Developer Reference for Intel® oneAPI Math Kernel Library for Fortran

Contents

Developer Reference for Intel® oneAPI Math Kernel Library - Fortran

1

For more documentation on this and other products, visit the oneAPI Documentation Library.

Intel® Math Kernel Library is now Intel® oneAPI Math Kernel Library (oneMKL).

Documentation for versions of Intel® Math Kernel Library older than 2023.0 is available for download only. See Downloadable Documentation.

What's New

C interface: *Developer Reference for Intel® oneAPI Math Kernel Library - C*

This publication describes the Fortran interface.

Basic Linear Algebra Subprograms (BLAS)	The BLAS routines provide vector, matrix-vector, and matrix-matrix operations.
Sparse BLAS	The Sparse BLAS routines provide basic operations on sparse vectors and matrices.
Sparse QR	The Sparse QR Routines provide a multifrontal sparse QR factorization method for solving a sparse system of linear equations.
LAPACK	The LAPACK routines solve systems of linear equations, least square problems, eigenvalue and singular value problems, and Sylvester's equations.
Statistical Functions	The Statistical Functions provides a set of routines implementing commonly used pseudorandom random number generators (RNG) with continuous distribution.
Direct and Iterative Sparse Solvers	Among several options for solving sparse linear systems of equations, oneMKL offers a direct sparse solver based on PARDISO*, which is referred to here as Intel MKL PARDISO.
Vector Mathematics Functions	The Vector Mathematics (VM) functions compute core mathematical functions on vector arguments.
Vector Statistics Functions	The Vector Statistics (VS) functions generate vectors of pseudorandom numbers with different types of statistical distributions and perform convolution and correlation computations.
Fourier Transform Functions	The Fourier Transform Functions offer several options for computing Fast Fourier Transforms (FFTs).

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Getting Help and Support

Intel provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more. Visit the Intel® oneAPI Math Kernel Library (oneMKL) support website athttp://www.intel.com/software/products/support/.

What's New

This Developer Reference documents Intel® oneAPI Math Kernel Library (oneMKL) release for the Fortran interface.

Intel® Math Kernel Library is now Intel® oneAPI Math Kernel Library (oneMKL). Documentation for older versions of Intel® Math Kernel Library is available for download only. For a list of available documentation downloads by product version, see these pages:

- Download Documentation for Intel® Parallel Studio XE
- Download Documentation for Intel® System Studio

The manual has been updated to reflect enhancements to the product, besides improvements and error corrections.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Notational Conventions

This manual uses the following terms to refer to operating systems:

Windows* OS	This term refers to information that is valid on all supported Windows* operating systems.
Linux* OS	This term refers to information that is valid on all supported Linux* operating systems.
macOS*	This term refers to information that is valid on Intel®-based systems running the macOS* operating system.

This manual uses the following notational conventions:

- Routine name shorthand (for example, `?ungqr` instead of `cungqr`/`zungqr`).
- Font conventions used for distinction between the text and the code.

Routine Name Shorthand

For shorthand, names that contain a question mark "`?`" represent groups of routines with similar functionality. Each group typically consists of routines used with four basic data types: single-precision real, double-precision real, single-precision complex, and double-precision complex. The question mark is used to indicate any or all possible varieties of a function; for example:

`?swap`	Refers to all four data types of the vector-vector `?swap` routine: `sswap`, `dswap`, `cswap`, and `zswap`.

Font Conventions

The following font conventions are used:

`UPPERCASE COURIER`	Data type used in the description of input and output parameters for Fortran interface. For example, `CHARACTER*1`.
`lowercase courier`	Code examples: `a(k+i,j) = matrix(i,j)`
`lowercase courier italic`	Variables in arguments and parameters description. For example, *incx*.
`*`	Used as a multiplication symbol in code examples and equations and where required by the programming language syntax.

Overview

NOTE
This publication, the *Intel® oneAPI Math Kernel Library Developer Reference*, was previously known as the *Intel® oneAPI Math Kernel Library Reference Manual*.

Intel® oneAPI Math Kernel Library (oneMKL) is optimized for performance on Intel processors. oneMKL also runs on non-Intel x86-compatible processors.

NOTE
oneMKL provides limited input validation to minimize the performance overheads. It is your responsibility when using oneMKL to ensure that input data has the required format and does not contain invalid characters. These can cause unexpected behavior of the library. Examples of the inputs that may result in unexpected behavior:

- Not-a-number (NaN) and other special floating point values
- Large inputs may lead to accumulator overflow

As the oneMKL API accepts raw pointers, it is your application's responsibility to validate the buffer sizes before passing them to the library. The library requires subroutine and function parameters to be valid before being passed. While some oneMKL routines do limited checking of parameter errors, your application should check for NULL pointers, for example.

The Intel® oneAPI Math Kernel Library includes Fortran routines and functions optimized for Intel® processor-based computers running operating systems that support multiprocessing. In addition to the Fortran interface, Intel® oneAPI Math Kernel Library (oneMKL) includes a C-language interface for the Discrete Fourier transform functions, as well as for the Vector Mathematics and Vector Statistics functions. For hardware and software requirements to use Intel® oneAPI Math Kernel Library (oneMKL), see*Intel® oneAPI Math Kernel Library (oneMKL) Release Notes*.

NOTE
Functions calls at runtime for Intel® oneAPI Math Kernel Library (oneMKL) libraries on the Microsoft Windows* operating system can utilize the function,`LoadLibrary()`, and related loading functions in static, dynamic, and single dynamic library linking models. These functions attempt to access the loader lock which when used within or at the same time as another `DllMain`function call, can lead to a deadlock. If possible, avoid making your calls to Intel® oneAPI Math Kernel Library (oneMKL) in a`DllMain` function or at the same time as other calls to `DllMain` even on separate threads. Refer to Microsoft documentation about *DllMain* and *Dynamic-Link Library Best Practices* for more details.

BLAS Routines

The BLAS routines and functions are divided into the following groups according to the operations they perform:

- BLAS Level 1 Routines perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- BLAS Level 2 Routines perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- BLAS Level 3 Routines perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

Starting from release 8.0, Intel® oneAPI Math Kernel Library (oneMKL) also supports the Fortran 95 interface to the BLAS routines.

Starting from release 10.1, a number of BLAS-like Extensions are added to enable the user to perform certain data manipulation, including matrix in-place and out-of-place transposition operations combined with simple matrix arithmetic operations.

Sparse BLAS Routines

The Sparse BLAS Level 1 Routines and Functions and Sparse BLAS Level 2 and Level 3 Routinesroutines and functions operate on sparse vectors and matrices. These routines perform vector operations similar to the BLAS Level 1, 2, and 3 routines. The Sparse BLAS routines take advantage of vector and matrix sparsity: they allow you to store only non-zero elements of vectors and matrices. Intel® oneAPI Math Kernel Library (oneMKL) also supports Fortran 95 interface to Sparse BLAS routines.

Sparse QR

Sparse QRin Intel® oneAPI Math Kernel Library (oneMKL) is a set of routines used to solve sparse matrices with real coefficients and general structure. All Sparse QR routines can be divided into three steps: reordering, factorization, and solving. Currently, only CSR format is supported for the input matrix, and Sparse QR operates on the matrix handle used in all SpBLAS IE routines. (For details on how to create a matrix handle, refer tomkl-sparse-create-csr.)

LAPACK Routines

The Intel® oneAPI Math Kernel Library fully supports the LAPACK 3.7 set of computational, driver, auxiliary and utility routines.

The original versions of LAPACK from which that part of Intel® oneAPI Math Kernel Library (oneMKL) was derived can be obtained fromhttp://www.netlib.org/lapack/index.html. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

The LAPACK routines can be divided into the following groups according to the operations they perform:

- Routines for solving systems of linear equations, factoring and inverting matrices, and estimating condition numbers (see LAPACK Routines: Linear Equations).
- Routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations (see LAPACK Routines: Least Squares and Eigenvalue Problems).
- Auxiliary and utility routines used to perform certain subtasks, common low-level computation or related tasks (see LAPACK Auxiliary Routines and LAPACK Utility Functions and Routines).

Starting from release 8.0, Intel® oneAPI Math Kernel Library (oneMKL) also supports the Fortran 95 interface to LAPACK computational and driver routines. This interface provides an opportunity for simplified calls of LAPACK routines with fewer required arguments.

ScaLAPACK Routines

The ScaLAPACK package (provided only for Intel® 64 architectures; see ScaLAPACK Routines) runs on distributed-memory architectures and includes routines for solving systems of linear equations, solving linear least squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

The original versions of ScaLAPACK from which that part of Intel® oneAPI Math Kernel Library (oneMKL) was derived can be obtained fromhttp://www.netlib.org/scalapack/index.html. The authors of ScaLAPACK are L. Blackford, J. Choi, A.Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K.Stanley, D. Walker, and R. Whaley.

The Intel® oneAPI Math Kernel Library (oneMKL) version of ScaLAPACK is optimized for Intel® processors and uses MPICH version of MPI as well as Intel MPI.

PBLAS Routines

The PBLAS routines perform operations with distributed vectors and matrices.

- PBLAS Level 1 Routines perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- PBLAS Level 2 Routines perform distributed matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- PBLAS Level 3 Routines perform distributed matrix-matrix operations, such as matrix-matrix multiplication, rank-*k* update, and solution of triangular systems.

Intel® oneAPI Math Kernel Library (oneMKL) provides the PBLAS routines with interface similar to the interface used in the Netlib PBLAS (part of the ScaLAPACK package, see http://www.netlib.org/scalapack/html/pblas_qref.html).

Sparse Solver Routines

Direct sparse solver routines in Intel® oneAPI Math Kernel Library (oneMKL) (seeSparse Solver Routines) solve symmetric and symmetrically-structured sparse matrices with real or complex coefficients. For symmetric matrices, these Intel® oneAPI Math Kernel Library (oneMKL) subroutines can solve both positive-definite and indefinite systems. Intel® oneAPI Math Kernel Library (oneMKL) includes a solver based on the PARDISO* sparse solver, referred to as Intel® oneAPI Math Kernel Library (oneMKL) PARDISO, as well as an alternative set of user callable direct sparse solver routines.

If you use the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO sparse solver, please cite:

O.Schenk and K.Gartner. Solving unsymmetric sparse systems of linear equations with PARDISO. J. of Future Generation Computer Systems, 20(3):475-487, 2004.

Intel® oneAPI Math Kernel Library (oneMKL) provides also an iterative sparse solver (seeSparse Solver Routines) that uses Sparse BLAS level 2 and 3 routines and works with different sparse data formats.

Extended Eigensolver Routines

TheExtended Eigensolver RCI Routines is a set of high-performance numerical routines for solving standard ($Ax = \lambda x$) and generalized ($Ax = \lambda Bx$) eigenvalue problems, where A and B are symmetric or Hermitian. It yields all the eigenvalues and eigenvectors within a given search interval. It is based on the Feast algorithm, an innovative fast and stable numerical algorithm presented in [Polizzi09], which deviates fundamentally from the traditional Krylov subspace iteration based techniques (Arnoldi and Lanczos algorithms [Bai00]) or other Davidson-Jacobi techniques [Sleijpen96]. The Feast algorithm is inspired by the density-matrix representation and contour integration technique in quantum mechanics.

It is free from orthogonalization procedures. Its main computational tasks consist of solving very few inner independent linear systems with multiple right-hand sides and one reduced eigenvalue problem orders of magnitude smaller than the original one. The Feast algorithm combines simplicity and efficiency and offers

many important capabilities for achieving high performance, robustness, accuracy, and scalability on parallel architectures. This algorithm is expected to significantly augment numerical performance in large-scale modern applications.

Some of the characteristics of the Feast algorithm [Polizzi09] are:

- Converges quickly in 2-3 iterations with very high accuracy
- Naturally captures all eigenvalue multiplicities
- No explicit orthogonalization procedure
- Can reuse the basis of pre-computed subspace as suitable initial guess for performing outer-refinement iterations

  This capability can also be used for solving a series of eigenvalue problems that are close one another.
- The number of internal iterations is independent of the size of the system and the number of eigenpairs in the search interval
- The inner linear systems can be solved either iteratively (even with modest relative residual error) or directly

VM Functions

The Vector Mathematics functions (see Vector Mathematical Functions) include a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic, etc.) that operate on vectors of real and complex numbers.

Application programs that might significantly improve performance with VM include nonlinear programming software, integrals computation, and many others. VM provides interfaces both for Fortran and C languages.

Statistical Functions

Vector Statistics (VS) contains three sets of functions (see Statistical Functions) providing:

- Pseudorandom, quasi-random, and non-deterministic random number generator subroutines implementing basic continuous and discrete distributions. To provide best performance, the VS subroutines use calls to highly optimized Basic Random Number Generators (BRNGs) and a set of vector mathematical functions.
- A wide variety of convolution and correlation operations.
- Initial statistical analysis of raw single and double precision multi-dimensional datasets.

Fourier Transform Functions

The Intel® oneAPI Math Kernel Library (oneMKL) multidimensional Fast Fourier Transform (FFT) functions with mixed radix support (see Fourier Transform Functions) provide uniformity of discrete Fourier transform computation and combine functionality with ease of use. Both Fortran and C interface specifications are given. There is also a cluster version of FFT functions, which runs on distributed-memory architectures and is provided only for Intel® 64 architectures.

The FFT functions provide fast computation via the FFT algorithms for arbitrary lengths. See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for the specific radices supported.

Partial Differential Equations Support

Intel® oneAPI Math Kernel Library (oneMKL) provides tools for solving Partial Differential Equations (PDE) (seePartial Differential Equations Support). These tools are Trigonometric Transform interface routines and Poisson Solver.

The Trigonometric Transform routines may be helpful to users who implement their own solvers similar to the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver. The users can improve performance of their solvers by using fast sine, cosine, and staggered cosine transforms implemented in the Trigonometric Transform interface.

The Poisson Solver is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The Trigonometric Transform interface, which underlies the solver, is based on the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (refer toFourier Transform Functions), optimized for Intel® processors.

Nonlinear Optimization Problem Solvers

Intel® oneAPI Math Kernel Library (oneMKL) provides Nonlinear Optimization Problem Solver routines (seeNonlinear Optimization Problem Solvers) that can be used to solve nonlinear least squares problems with or without linear (bound) constraints through the Trust-Region (TR) algorithms and compute Jacobi matrix by central differences.

Support Functions

The Intel® oneAPI Math Kernel Library (oneMKL) support functions (seeSupport Functions) are used to support the operation of the Intel® oneAPI Math Kernel Library (oneMKL) software and provide basic information on the library and library operation, such as the current library version, timing, setting and measuring of CPU frequency, error handling, and memory allocation.

Starting from release 10.0, the Intel® oneAPI Math Kernel Library (oneMKL) support functions provide additional threading control.

Starting from release 10.1, Intel® oneAPI Math Kernel Library (oneMKL) selectively supports a*Progress Routine* feature to track progress of a lengthy computation and/or interrupt the computation using a callback function mechanism. The user application can define a function called `mkl_progress`that is regularly called from the Intel® oneAPI Math Kernel Library (oneMKL) routine supporting the progress routine feature. SeeProgress Routine in Support Functions for reference. Refer to a specific LAPACK or DSS/PARDISO function description to see whether the function supports this feature or not.

BLACS Routines

The Intel® oneAPI Math Kernel Library implements routines from the BLACS (Basic Linear Algebra Communication Subprograms) package (seeBLACS Routines) that are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The original versions of BLACS from which that part of Intel® oneAPI Math Kernel Library (oneMKL) was derived can be obtained from http://www.netlib.org/blacs/index.html. The authors of BLACS are Jack Dongarra and R. Clint Whaley.

Data Fitting Functions

The Data Fitting component includes a set of highly-optimized implementations of algorithms for the following spline-based computations:

- spline construction
- interpolation including computation of derivatives and integration
- search

The algorithms operate on single and double vector-valued functions set in the points of the given partition. You can use Data Fitting algorithms in applications that are based on data approximation. See Data Fitting Functions for more information.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Performance Enhancements

The Intel® oneAPI Math Kernel Library has been optimized by exploiting both processor and system features and capabilities. Special care has been given to those routines that most profit from cache-management techniques. These especially include matrix-matrix operation routines such as`dgemm()`.

In addition, code optimization techniques have been applied to minimize dependencies of scheduling integer and floating-point units on the results within the processor.

The major optimization techniques used throughout the library include:

- Loop unrolling to minimize loop management costs
- Blocking of data to improve data reuse opportunities
- Copying to reduce chances of data eviction from cache
- Data prefetching to help hide memory latency
- Multiple simultaneous operations (for example, dot products in `dgemm`) to eliminate stalls due to arithmetic unit pipelines
- Use of hardware features such as the SIMD arithmetic units, where appropriate

These are techniques from which the arithmetic code benefits the most.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Parallelism

Intel® oneAPI Math Kernel Library (oneMKL) offers performance gains through parallelism provided by the symmetric multiprocessing performance (SMP) feature. You can obtain improvements from SMP in the following ways:

- One way is based on user-managed threads in the program and further distribution of the operations over the threads based on data decomposition, domain decomposition, control decomposition, or some other parallelizing technique. Each thread can use any of the Intel® oneAPI Math Kernel Library (oneMKL) functions (except for the deprecated`?lacon` LAPACK routine) because the library has been designed to be thread-safe.
- Another method is to use the FFT and BLAS level 3 routines. They have been parallelized and require no alterations of your application to gain the performance enhancements of multiprocessing. Performance using multiple processors on the level 3 BLAS shows excellent scaling. Since the threads are called and managed within the library, the application does not need to be recompiled thread-safe (see also Fortran 95 Interface Conventions in BLAS and Sparse BLAS Routines).
- Yet another method is to use *tuned LAPACK routines*. Currently these include the single- and double precision flavors of routines for *QR* factorization of general matrices, triangular factorization of general and symmetric positive-definite matrices, solving systems of equations with such matrices, as well as solving symmetric eigenvalue problems.

For instructions on setting the number of available processors for the BLAS level 3 and LAPACK routines, see *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

OpenMP* Offload

This section describes how to perform OpenMP offload computations using Intel® oneAPI Math Kernel Library.

OpenMP* Offload for Intel® oneAPI Math Kernel Library

You can use Intel® oneAPI Math Kernel Library (oneMKL) and OpenMP* offload to run standard oneMKL computations on Intel GPUs. You can find the list of oneMKL features that support OpenMP offload in the `mkl_omp_offload.f90` interface module file which includes:

- All Level 1, 2, and 3 BLAS functions, supporting both synchronous and asynchronous execution
- BLAS-like extensions: `?axpby`, `?axpy_batch_strided`, `?gemv_batch_strided`, `?dgmm_batch_strided`, `gemm_s8u8s32`, `?gemm_batch_strided`, `?trsm_batch_strided`, `?gemmt`, `mkl_?omatcopy_batch_strided`, `mkl_?imatcopy_batch_strided`, and `mkl_?omatadd_batch_strided`, supporting synchronous and asynchronous execution
- LAPACK, including LAPACK-like extensions
  - All computations on the Intel GPU (supports both synchronous and asynchronous execution):
    - `?getrf_batch_strided`
    - `?getrfnp_batch_strided`
    - `?getri`
    - `?getri_oop_batch_strided`
    - `?getrs`
    - `?getrs_batch_strided`
    - `?getrsnp_batch_strided`
    - `?gels_batch_strided`
    - `?potrf`
    - `?potri`
    - `?potrs`
    - `?trtri`
    - `?trtrs`
  - Hybrid; some computations on the Intel GPU (supports synchronous execution):
    - `?geqrf`
    - `?getrf` (all computations on the CPU for n <= 256)
    - `mkl_?getrfnp` (all computations on the CPU for n <= 512)
    - `?ormqr`, `?unmqr`
  - Interface support only; all computations on the CPU (supports synchronous execution):
    - `?gebrd`
    - `?gesvd`
    - `?orgqr`, `?ungqr`
    - `?steqr`
    - `?syev`, `?heev`
    - `?syevd`, `?heevd`
    - `?syevx`, `?heevx`
    - `?sygvd`, `?hegvd`
    - `?sygvx`, `?hegvx`
    - `?sytrd`, `?hetrd`
- FFTs through both DFTI and FFTW3 interfaces in one, two, and three dimensions.
  - For COMPLEX_STORAGE, only the DFTI_COMPLEX_COMPLEX format is currently supported on CPU and GPU devices.
  - Both synchronous and asynchronous computations are supported.
  - For R2C/C2R transforms on the GPU, only DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX is supported (implying DFTI_PACKED_FORMAT=DFTI_CCE_FORMAT).

- 2D and 3D FFTs are supported using rank-2/rank-3 Fortran arrays for input and output only through Fortran OpenMP offload interface.
- **NOTE** `INCONSISTENT_CONFIGURATION` errors at compute time indicate an invalid descriptor or invalid data pointer. Double check your data mapping if you encounter such errors.

- Arbitrary strides and batch distances are not supported for multi-dimensional R2C transforms offloaded to the GPU. Considering the first dimension of the data, every element must be separated from its two nearest peers (along another dimension and/or in another batch) by a constant distance. For example, to compute a batched, two-dimensional R2C FFT of size `[N1, N2]` with input strides `[0, 1, S2]` (column-major layout with unit elementary stride and no offset), `INPUT_DISTANCE` must be equal to `N2*S2` so that every element is separated from its nearest first-dimension counterpart(s) by a distance `S2` (in this example), even across batches.
- Transforms on GPU devices may overwrite FFT-irrelevant, padding entries in the output data.

- Vector Statistics
  - Random number generators

    NOTE
    All distributions are supported. See https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-fortran/current/distribution-generators.html

    Basic random number generators:

    - VSL_BRNG_MCG31
    - VSL_BRNG_MCG59
    - VSL_BRNG_PHILOX4X32X10
    - VSL_BRNG_MRG32K3A
    - VSL_BRNG_MT19937
    - VSL_BRNG_MT2203
    - VSL_BRNG_SOBOL
  - Summary statistics

    Supports the `vsl?SSCompute` routine for the following estimates:

    - VSL_SS_MEAN
    - VSL_SS_SUM
    - VSL_SS_2R_MOM
    - VSL_SS_2R_SUM
    - VSL_SS_3R_MOM
    - VSL_SS_3R_SUM
    - VSL_SS_4R_MOM
    - VSL_SS_4R_SUM
    - VSL_SS_2C_MOM
    - VSL_SS_2C_SUM
    - VSL_SS_3C_MOM
    - VSL_SS_3C_SUM
    - VSL_SS_4C_MOM
    - VSL_SS_4C_SUM
    - VSL_SS_KURTOSIS
    - VSL_SS_SKEWNESS
    - VSL_SS_MIN
    - VSL_SS_MAX
    - VSL_SS_VARIATION

    Supported methods:

    - VSL_SS_METHOD_FAST
    - VSL_SS_METHOD_FAST_USER_MEAN

The OpenMP offload feature from Intel® oneAPI Math Kernel Library (oneMKL) allows you to run oneMKL computations on Intel GPUs through the standard oneMKL APIs within an `omp target variant dispatch` section. For example, the standard CBLAS API for single precision real data type matrix multiply is:

```
subroutine sgemm ( transa, transb, m, n, k, alpha, a, lda,        &
          &b, ldb, beta, c, ldc ) BIND(C)
       character*1,intent(in)              :: transa, transb
       integer,intent(in)                  :: m, n, k, lda, ldb, ldc
       real,intent(in)                     :: alpha, beta
       real,intent(in)                     :: a( lda, * ), b( ldb, * )
       real,intent(inout)                  :: c( ldc, * )
     end subroutine sgemm
```

If `sgemm` is called outside of an `omp target variant dispatch` section or if offload is disabled, then the CPU implementation is dispatched. If the same function is called within an `omp target variant dispatch` section and offload is possible then the GPU implementation is dispatched. By default the execution of the oneMKL function within a dispatch variant construct is synchronous, the `nowait` clause can be used on the `dispatch variant` construct to specify that asynchronous execution is desired. In that case, synchronization needs to be handled by the application using standard OpenMP synchronization functionality, for example the `omp taskwait` construct.

In order to offload to a device, arguments to the oneMKL function must be mapped to the device memory if they represent a return value (marked with `intent(out)` or `intent(inout)` in the subroutine declaration) or if they point to an array of data (such as a matrix or vector, even if it is an input array). Users must map these arguments to the device using the `omp target data` construct before calling the oneMKL routine, and in addition must use the `use_device_ptr` clause in the `omp target variant dispatch` construct to specify variables that have been mapped to device memory.

In Fortran, the OpenMP Offload interfaces have stricter type checking than the standard Fortran interfaces for the same functions. For BLAS functions and BLAS-like extensions, you can bypass this stricter type checking by changing the module that is loaded. For example, in the example below, include `use onemkl_blas_omp_offload_lp64_no_array_check` instead of `use onemkl_blas_omp_offload_lp64`.

Example

Examples for using the OpenMP offload for oneMKL are located in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory, under:

```
examples/f_offload
```

```
include "mkl_omp_offload.f90"

program sgemm_example
use onemkl_blas_omp_offload_lp64
use common_blas

character*1 :: transa = 'N', transb = 'N'
integer :: i, j, m = 5, n = 3, k = 10
integer :: lda, ldb, ldc
real :: alpha = 1.0, beta = 1.0
real,allocatable :: a(:,:), b(:,:), c(:,:)

! initialize leading dimension and allocate and initialize arrays
lda = m
…
allocate(a(lda,k))
…

! initialize matrices
call sinit_matrix(transa, m, k, lda, a)
…
```

```
! Calling sgemm on the CPU using standard oneMKL Fortran interface
call sgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

! map the a, b and c matrices on the device memory
!$omp target data map(a,b,c)

! Calling sgemm on the GPU using standard oneMKL Fortran interface within a variant dispatch
construct
! Use the use_device_ptr clause to specify that a, b and c are device memory
!$omp target variant dispatch use_device_ptr(a,b,c)
call sgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
!$omp end target variant dispatch

!$omp end target data

! Free memory
deallocate(a)
…
stop
end program
```

BLAS and Sparse BLAS Routines

Intel® oneAPI Math Kernel Library (oneMKL)implements the BLAS and Sparse BLAS routines, and BLAS-like extensions. The routine descriptions are arranged in several sections:

- BLAS Level 1 Routines (vector-vector operations)
- BLAS Level 2 Routines (matrix-vector operations)
- BLAS Level 3 Routines (matrix-matrix operations)
- Sparse BLAS Level 1 Routines (vector-vector operations).
- Sparse BLAS Level 2 and Level 3 Routines (matrix-vector and matrix-matrix operations)
- BLAS-like Extensions

The question mark in the group name corresponds to different character codes indicating the data type (`s`, `d`, `c`, and `z` or their combination); see *Routine Naming Conventions*.

When BLAS or Sparse BLAS routines encounter an error, they call the error reporting routine `xerbla`.

BLAS Routines

Naming Conventions for BLAS Routines

BLAS routine names have the following structure:

<character> <name> <mod>

The *<character>* field indicates the data type:

`s`	real, single precision
`c`	complex, single precision
`d`	real, double precision
`z`	complex, double precision

Some routines and functions can have combined character codes, such as `sc` or `dz`.

For example, the function `scasum` uses a complex input array and returns a real value.

The <*name*> field, in BLAS level 1, indicates the operation type. For example, the BLAS level 1 routines `?dot`, `?rot`, `?swap` compute a vector dot product, vector rotation, and vector swap, respectively.

In BLAS level 2 and 3, <*name*> reflects the matrix argument type:

`ge`	general matrix
`gb`	general band matrix
`sy`	symmetric matrix
`sp`	symmetric matrix (packed storage)
`sb`	symmetric band matrix
`he`	Hermitian matrix
`hp`	Hermitian matrix (packed storage)
`hb`	Hermitian band matrix
`tr`	triangular matrix
`tp`	triangular matrix (packed storage)
`tb`	triangular band matrix.

The <*mod*> field, if present, provides additional details of the operation. BLAS level 1 names can have the following characters in the <*mod*> field:

`c`	conjugated vector
`u`	unconjugated vector
`g`	Givens rotation construction
`m`	modified Givens rotation
`mg`	modified Givens rotation construction

BLAS level 2 names can have the following characters in the <*mod*> field:

`mv`	matrix-vector product
`sv`	solving a system of linear equations with a single unknown vector
`r`	rank-1 update of a matrix
`r2`	rank-2 update of a matrix.

BLAS level 3 names can have the following characters in the <*mod*> field:

`mm`	matrix-matrix product
`sm`	solving a system of linear equations with multiple unknown vectors
`rk`	rank-*k* update of a matrix
`r2k`	rank-2*k* update of a matrix.

The examples below illustrate how to interpret BLAS routine names:

`ddot`	`<d> <dot>`: real and double precision, vector-vector dot product
`cdotc`	`<c> <dot> <c>`: complex and single precision, vector-vector dot product, conjugated

`cdotu`	`<c> <dot> <u>`: complex and single precision, vector-vector dot product, unconjugated
`scasum`	`<sc> <asum>`: real and single-precision output, complex and single-precision input, sum of magnitudes of vector elements
`sgemv`	`<s> <ge> <mv>`: real and single precision, general matrix, matrix-vector product
`ztrmm`	`<z> <tr> <mm>`: complex and double precision, triangular matrix, matrix-matrix product

Sparse BLAS level 1 naming conventions are similar to those of BLAS level 1. For more information, see Naming Conventions.

Fortran 95 Interface Conventions for BLAS Routines

The Fortran 95 interface to BLAS and Sparse BLAS Level 1 routines is implemented through wrappers that call respective FORTRAN 77 routines. This interface uses features of Fortran 95 such as assumed-shape arrays and optional arguments to provide simplified calls to BLAS and Sparse BLAS Level 1 routines with fewer parameters.

For BLAS, Intel® oneAPI Math Kernel Library (oneMKL) offers two types of Fortran 95 interfaces:

- using `mkl_blas.fi` only through `include 'mkl.fi'` statement. Such interfaces allow you to make use of the original BLAS routines with all their arguments
- using `blas.f90` that includes improved interfaces. This file is used to generate the module files `blas95.mod` and `f95_precision.mod`. For details, see Fortran 95 interfaces and wrappers to LAPACK and BLAS. The module files are used to process the FORTRAN use clauses referencing the BLAS interface: `use blas95` and `use f95_precision`.

The main conventions used in Fortran 95 interface are as follows:

- The names of parameters used in Fortran 95 interface are typically the same as those used for the respective generic (FORTRAN 77) interface. In rare cases formal argument names may be different.
- Some input parameters such as array dimensions are not required in Fortran 95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.
- A parameter can be skipped if its value is completely defined by the presence or absence of another parameter in the calling sequence, and the restored value is the only meaningful value for the skipped parameter.
- Parameters specifying the increment values *incx* and *incy* are skipped. In most cases their values are equal to 1. In Fortran 95 an increment with different value can be directly established in the corresponding parameter.
- Some generic parameters are declared as optional in Fortran 95 interface and may or may not be present in the calling sequence. A parameter can be declared optional if it satisfies one of the following conditions:

  1. It can take only a few possible values. The default value of such parameter typically is the first value in the list; all exceptions to this rule are explicitly stated in the routine description.
  2. It has a natural default value.

  Optional parameters are given in square brackets in Fortran 95 call syntax.

The particular rules used for reconstructing the values of omitted optional parameters are specific for each routine and are detailed in the respective "Fortran 95 Notes" subsection at the end of routine specification section. If this subsection is omitted, the Fortran 95 interface for the given routine does not differ from the corresponding FORTRAN 77 interface.

Note that this interface is not implemented in the current version of Sparse BLAS Level 2 and Level 3 routines.

Matrix Storage Schemes for BLAS Routines

Matrix arguments of BLAS routines can use the following storage schemes:

- *Full storage*: a matrix *A* is stored in a two-dimensional array *a*, with the matrix element A_{ij} stored in the array element `a(i,j)`.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: a band matrix is stored compactly in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

For more information on matrix storage schemes, see Matrix Arguments in the Appendix "Routine and Function Arguments".

BLAS Level 1 Routines and Functions

BLAS Level 1 includes routines and functions, which perform vector-vector operations. The following table lists the BLAS Level 1 routine and function groups and the data types associated with them.

BLAS Level 1 Routine and Function Groups and Their Data Types

Routine or Function Group	Data Types	Description
?asum	s, d, sc, dz	Sum of vector magnitudes (functions)
?axpy	s, d, c, z	Scalar-vector product (routines)
?copy	s, d, c, z	Copy vector (routines)
?dot	s, d	Dot product (functions)
?sdot	sd, d	Dot product with double precision (functions)
?dotc	c, z	Dot product conjugated (functions)
?dotu	c, z	Dot product unconjugated (functions)
?nrm2	s, d, sc, dz	Vector 2-norm (Euclidean norm) (functions)
?rot	s, d, c, z, cs, zd	Plane rotation of points (routines)
?rotg	s, d, c, z	Generate Givens rotation of points (routines)
?rotm	s, d	Modified Givens plane rotation of points (routines)
?rotmg	s, d	Generate modified Givens plane rotation of points (routines)
?scal	s, d, c, z, cs, zd	Vector-scalar product (routines)
?swap	s, d, c, z	Vector-vector swap (routines)
i?amax	s, d, c, z	Index of the maximum absolute value element of a vector (functions)
i?amin	s, d, c, z	Index of the minimum absolute value element of a vector (functions)
?cabs1	s, d	Auxiliary functions, compute the absolute value of a complex number of single or double precision

?asum

Computes the sum of magnitudes of the vector elements.

Syntax

res = sasum(*n*, *x*, *incx*)

res = scasum(*n*, *x*, *incx*)

res = dasum(*n*, *x*, *incx*)

res = dzasum(*n*, *x*, *incx*)

res = asum(*x*)

Include Files

- mkl.fi, blas.f90

Description

The ?asum routine computes the sum of the magnitudes of elements of a real vector, or the sum of magnitudes of the real and imaginary parts of elements of a complex vector:

res = |Rex_1| + |Imx_1| + |Rex_2| + Imx_2|+ ... + |Rex_n| + |Imx_n|,

$$result = \sum_{i=1}^{n} \left(\left| \mathrm{Re}\left(X_i\right) \right| + \left| Im\left(X_i\right) \right| \right)$$

where *x* is a vector with *n* elements.

Input Parameters

n	INTEGER. Specifies the number of elements in vector *x*.
x	REAL for sasum DOUBLE PRECISION for dasum COMPLEX for scasum DOUBLE COMPLEX for dzasum Array, size at least (1 + (*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for indexing vector *x*.

Output Parameters

res	REAL for sasum DOUBLE PRECISION for dasum REAL for scasum DOUBLE PRECISION for dzasum Contains the sum of magnitudes of real and imaginary parts of all elements of the vector.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `asum` interface are the following:

x	Holds the array of size *n*.

?axpy

Computes a vector-scalar product and adds the result to a vector.

Syntax

```
call saxpy(n, a, x, incx, y, incy)
call daxpy(n, a, x, incx, y, incy)
call caxpy(n, a, x, incx, y, incy)
call zaxpy(n, a, x, incx, y, incy)
call axpy(x, y [,a])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?axpy` routines perform a vector-vector operation defined as

```
y := a*x + y
```

where:

a is a scalar

x and *y* are vectors each with a number of elements that equals *n*.

Input Parameters

n	`INTEGER`. Specifies the number of elements in vectors *x* and *y*.

a	`REAL` for `saxpy`
	`DOUBLE PRECISION` for `daxpy`
	`COMPLEX` for `caxpy`
	`DOUBLE COMPLEX` for `zaxpy`
	Specifies the scalar *a*.

x	`REAL` for `saxpy`
	`DOUBLE PRECISION` for `daxpy`
	`COMPLEX` for `caxpy`
	`DOUBLE COMPLEX` for `zaxpy`
	Array, size at least `(1 + (n-1)*abs(incx))`.

incx	`INTEGER`. Specifies the increment for the elements of *x*.
y	`REAL` for `saxpy`

	DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Array, size at least (1 + (*n*-1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

y	Contains the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine axpy interface are the following:

x	Holds the array of size *n*.
y	Holds the array of size *n*.
a	The default value is 1.

?copy

Copies a vector to another vector.

Syntax

```
call scopy(n, x, incx, y, incy)
call dcopy(n, x, incx, y, incy)
call ccopy(n, x, incx, y, incy)
call zcopy(n, x, incx, y, incy)
call copy(x, y)
```

Include Files

- mkl.fi, blas.f90

Description

The ?copy routines perform a vector-vector operation defined as

```
y = x,
```

where *x* and *y* are vectors.

Input Parameters

n	INTEGER. Specifies the number of elements in vectors *x* and *y*.
x	REAL for scopy

	DOUBLE PRECISION for dcopy COMPLEX for ccopy DOUBLE COMPLEX for zcopy Array, size at least (1 + (*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for scopy DOUBLE PRECISION for dcopy COMPLEX for ccopy DOUBLE COMPLEX for zcopy Array, size at least (1 + (*n*-1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

y	Contains a copy of the vector *x* if *n* is positive. Otherwise, parameters are unaltered.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine copy interface are the following:

x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.

?copy_batch

Computes a group of vector copies.

Syntax

```
call scopy_batch(n_array, x_array, incx_array, y_array, incy_array, group_count, group_size_array)
call dcopy_batch(n_array, x_array, incx_array, y_array, incy_array, group_count, group_size_array)
call ccopy_batch(n_array, x_array, incx_array, y_array, incy_array, group_count, group_size_array)
call zcopy_batch(n_array, x_array, incx_array, y_array, incy_array, group_count, group_size_array)
```

Description

The ?copy_batch routines perform a series of vector copies. They are similar to their ?copy routine counterparts, but the ?copy_batch routines perform vector operations with groups of vectors. Each groups contains vectors with the same parameters (size, incrmenet), but the parameters can vary between groups.

The operation is defined as follows:

```
idx = 0
for i = 0 … group_count – 1
    n, incx, incy and group_size at position i in n_array, incx_array, incy_array and
group_size_array
    for j = 0 … group_size – 1
        x and y are vectors of size n at position idx in x_array and y_array
        y := x
        idx := idx + 1
    end for
end for
```

The number of entries in `x_array` and `y_array` is `total_batch_count`, which is the sum of all the `group_size` entries.

Input Parameters

n_array	INTEGER. Array of size `group_count`. For the group `i`, `n_i = n_array[i]` is the number of elements in the vectors `x` and `y`.
x_array	INTEGER*8 for Intel® 64 architecture INTEGER*4 for IA-32 architecture Array of size `total_batch_count` of pointers used to store `x` vectors. The array allocated for the `x` vectors of the group `i` must be of size at least `(1 + (n_i - 1)*abs(incx_i))`.
incx_array	INTEGER. Array of size `group_count`. For the group `i`, `incx_i = incx_array[i]` is the distance between consecutive entries in a vector `x`.
y_array	INTEGER*8 for Intel® 64 architecture INTEGER*4 for IA-32 architecture Array of size `total_batch_count` of pointers used to store `y` vectors. The array allocated for the `y` vectors of the group `i` must be of size at least `(1 + (n_i - 1)*abs(incy_i))`.
incy_array	INTEGER. Array of size `group_count`. For the group `i`, `incy_i = incy_array[i]` is the distance between consecutive entries in a vector `y`.
group_count	INTEGER. Number of groups. Must be at least 0.
group_size_array	INTEGER. Array of size `group_count`. The element `group_size_array[i]` is the number of vector in the group i. Each element in group_size_array must be at least 0.

Output Parameters

y_array	Array of pointers holding the `total_batch_count` copied vectors `y`.

?copy_batch_strided

Computes a group of vector copies.

Syntax

```
call scopy_batch_strided(n, x, incx, stridex, y, incy, stridey, batch_size)
call dcopy_batch_strided(n, x, incx, stridex, y, incy, stridey, batch_size)
```

```
call ccopy_batch_strided(n, x, incx, stridex, y, incy, stridey, batch_size)
call zcopy_batch_strided(n, x, incx, stridex, y, incy, stridey, batch_size)
```

Description

The ?copy_batch_strided routines perform a series of vector copies. They are similar to their ?copy routine counterparts, but the ?copy_batch_strided routines perform vector operations with a group of vectors.

All vectors x (respectively, y) have the same parameters (size, increment) and are stored at constant distance stridex (respectively, stridey) from each other. The operation is defined as follows:

```
for i = 0 … batch_size - 1
    X and Y are vectors at offset i * stridex and i * stridey in x and y
    Y = X
end for
```

Input Parameters

n	INTEGER. Number of elements in vectors x and y.
x	REAL for scopy_batch_strided DOUBLE PRECISION for dcopy_batch_strided COMPLEX for ccopy_batch_strided DOUBLE COMPLEX for zcopy_batch_strided Array of size at least stridex*batch_size holding the input x vectors.
incx	INTEGER. Specifies the increment between two consecutive elements of a single vector x.
stridex	INTEGER. Stride between two consecutive x vectors. Must be at least (1 + (n-1)*abs(incx)).
y	REAL for scopy_batch_strided DOUBLE PRECISION for dcopy_batch_strided COMPLEX for ccopy_batch_strided DOUBLE COMPLEX for zcopy_batch_strided Array of size at least stridey*batch_size holding the output y vectors.
incy	INTEGER. Specifies the increment between two consecutive elements of a single vector y.
stridey	INTEGER. Stride between two consecutive y vectors. Must be at least (1 + (n-1)*abs(incy)).
batch_size	INTEGER. Number of copy computations to perform and x and y vectors. Must be at least 0.

Output Parameters

y	Array holding the batch_size copied vectors y.

?dot

Computes a vector-vector dot product.

Syntax

res = sdot(*n*, *x*, *incx*, *y*, *incy*)

res = ddot(*n*, *x*, *incx*, *y*, *incy*)

res = dot(*x*, *y*)

Include Files

- mkl.fi, blas.f90

Description

The ?dot routines perform a vector-vector reduction operation defined as

$$res = \sum_{i=1}^{n} x_i * y_i,$$

where x_i and y_i are elements of vectors *x* and *y*.

Input Parameters

n	INTEGER. Specifies the number of elements in vectors *x* and *y*.
x	REAL for sdot DOUBLE PRECISION for ddot Array, size at least (1+(*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for sdot DOUBLE PRECISION for ddot Array, size at least (1+(*n*-1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

res	REAL for sdot DOUBLE PRECISION for ddot Contains the result of the dot product of *x* and *y*, if *n* is positive. Otherwise, *res* contains 0.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `dot` interface are the following:

x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.

?sdot

Computes a vector-vector dot product with double precision.

Syntax

```
res = sdsdot(n, sb, sx, incx, sy, incy)
res = dsdot(n, sx, incx, sy, incy)
res = sdot(sx, sy)
res = sdot(sx, sy, sb)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?sdot` routines compute the inner product of two vectors with double precision. Both routines use double precision accumulation of the intermediate results, but the `sdsdot` routine outputs the final result in single precision, whereas the `dsdot` routine outputs the double precision result. The function `sdsdot` also adds scalar value *sb* to the inner product.

Input Parameters

n	`INTEGER`. Specifies the number of elements in the input vectors *sx* and *sy*.
sb	`REAL`. Single precision scalar to be added to inner product (for the function `sdsdot` only).
sx, *sy*	`REAL`. Arrays, size at least `(1+(n -1)*abs(incx))` and `(1+(n-1)*abs(incy))`, respectively. Contain the input single precision vectors.
incx	`INTEGER`. Specifies the increment for the elements of *sx*.
incy	`INTEGER`. Specifies the increment for the elements of *sy*.

Output Parameters

res	`REAL` for `sdsdot` `DOUBLE PRECISION` for `dsdot`

Contains the result of the dot product of *sx* and *sy* (with *sb* added for `sdsdot`), if *n* is positive. Otherwise, *res* contains *sb* for `sdsdot` and `0` for `dsdot`.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `sdot` interface are the following:

sx Holds the vector with the number of elements *n*.

sy Holds the vector with the number of elements *n*.

NOTE
Note that scalar parameter *sb* is declared as a required parameter in Fortran 95 interface for the function `sdot` to distinguish between function flavors that output final result in different precision.

?dotc

Computes a dot product of a conjugated vector with another vector.

Syntax

```
res = cdotc(n, x, incx, y, incy)
res = zdotc(n, x, incx, y, incy)
res = dotc(x, y)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?dotc` routines perform a vector-vector operation defined as:

$$res = \sum_{i=1}^{n} conjg(x_i) * y_i,$$

where x_i and y_i are elements of vectors *x* and *y*.

Input Parameters

n `INTEGER`. Specifies the number of elements in vectors *x* and *y*.

x `COMPLEX` for `cdotc`

`DOUBLE COMPLEX` for `zdotc`

	Array, size at least `(1 + (n -1)*abs(incx))`.
incx	`INTEGER`. Specifies the increment for the elements of *x*.

y	`COMPLEX` for `cdotc`
	`DOUBLE COMPLEX` for `zdotc`
	Array, size at least `(1 + (n -1)*abs(incy))`.

incy	`INTEGER`. Specifies the increment for the elements of *y*.

Output Parameters

res	`COMPLEX` for `cdotc`
	`DOUBLE COMPLEX` for `zdotc`
	Contains the result of the dot product of the conjugated *x* and unconjugated *y*, if *n* is positive. Otherwise, it contains `0`.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `dotc` interface are the following:

x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.

?dotu

Computes a complex vector-vector dot product.

Syntax

```
res = cdotu(n, x, incx, y, incy)
res = zdotu(n, x, incx, y, incy)
res = dotu(x, y)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?dotu` routines perform a vector-vector reduction operation defined as

$$res = \sum_{i=1}^{n} x_i * y_i$$

where x_i and y_i are elements of complex vectors *x* and *y*.

Input Parameters

n	INTEGER. Specifies the number of elements in vectors *x* and *y*.
x	COMPLEX for cdotu DOUBLE COMPLEX for zdotu Array, size at least (1 + (*n* -1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	COMPLEX for cdotu DOUBLE COMPLEX for zdotu Array, size at least (1 + (*n* -1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

res	COMPLEX for cdotu DOUBLE COMPLEX for zdotu Contains the result of the dot product of *x* and *y*, if *n* is positive. Otherwise, it contains 0.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine dotu interface are the following:

x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.

?nrm2

Computes the Euclidean norm of a vector.

Syntax

```
res = snrm2(n, x, incx)
res = dnrm2(n, x, incx)
res = scnrm2(n, x, incx)
res = dznrm2(n, x, incx)
res = nrm2(x)
```

Include Files

- mkl.fi, blas.f90

Description

The ?nrm2 routines perform a vector reduction operation defined as

```
res = ||x||,
```

where:

x is a vector,

res is a value containing the Euclidean norm of the elements of *x*.

Input Parameters

n	INTEGER. Specifies the number of elements in vector *x*.
x	REAL for snrm2 DOUBLE PRECISION for dnrm2 COMPLEX for scnrm2 DOUBLE COMPLEX for dznrm2 Array, size at least (1 + (*n* -1)*abs (*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

res	REAL for snrm2 DOUBLE PRECISION for dnrm2 REAL for scnrm2 DOUBLE PRECISION for dznrm2 Contains the Euclidean norm of the vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine nrm2 interface are the following:

x	Holds the vector with the number of elements *n*.

?rot

Performs rotation of points in the plane.

Syntax

```
call srot(n, x, incx, y, incy, c, s)
call drot(n, x, incx, y, incy, c, s)
call crot(n, x, incx, y, incy, c, s)
call zrot(n, x, incx, y, incy, c, s)
call csrot(n, x, incx, y, incy, c, s)
call zdrot(n, x, incx, y, incy, c, s)
call rot(x, y, c, s)
```

Description

Given two complex vectors *x* and *y*, each vector element of these vectors is replaced as follows:

```
xi = c*xi + s*yi
yi = c*yi - s*xi
```

If *s* is a complex type, each vector element is replaced as follows:

```
xi = c*xi + s*yi
yi = c*yi - conj(s)*xi
```

Input Parameters

n	INTEGER. Specifies the number of elements in vectors *x* and *y*.
x	REAL for srot DOUBLE PRECISION for drot COMPLEX for csrot DOUBLE COMPLEX for zdrot Array, size at least (1 + (*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for srot DOUBLE PRECISION for drot COMPLEX for csrot DOUBLE COMPLEX for zdrot Array, size at least (1 + (*n* -1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.
c	REAL for srot DOUBLE PRECISION for drot REAL for csrot DOUBLE PRECISION for zdrot A scalar.
s	REAL for srot DOUBLE PRECISION for drot COMPLEX for crot DOUBLE COMPLEX for zrot REAL for csrot DOUBLE PRECISION for zdrot A scalar.

Output Parameters

x	Each element is replaced by *c*x + s*y*.

y	Each element is replaced by `c*y - s*x`, or by `c*y-conj(s)*x` if *s* is a complex type.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `rot` interface are the following:

x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.

?rotg

Computes the parameters for a Givens rotation.

Syntax

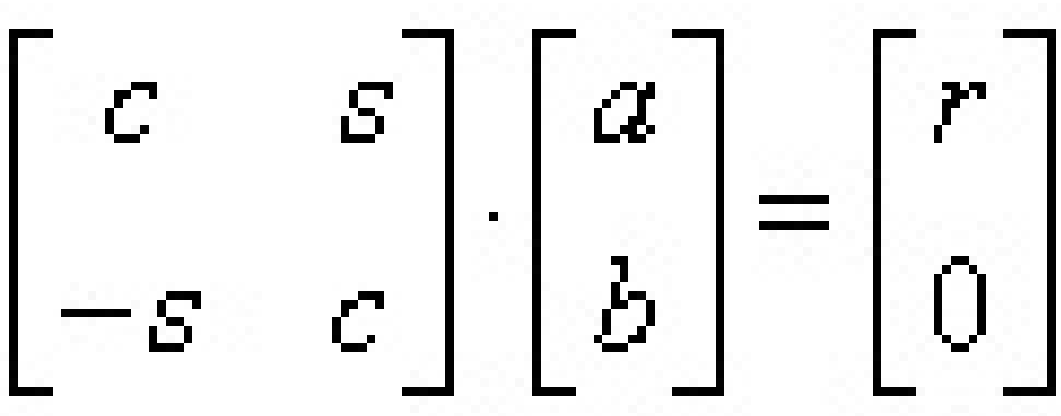
```
call srotg(a, b, c, s)
call drotg(a, b, c, s)
call crotg(a, b, c, s)
call zrotg(a, b, c, s)
call rotg(a, b, c, s)
```

Include Files

- `mkl.fi, blas.f90`

Description

Given the Cartesian coordinates `(a, b)` of a point, these routines return the parameters *c*, *s*, *r*, and *z* associated with the Givens rotation. The parameters *c* and *s* define a unitary matrix such that:



The parameter *z* is defined such that if $|a| > |b|$, *z* is *s*; otherwise if *c* is not 0 *z* is $1/c$; otherwise *z* is 1.

See a more accurate LAPACK version `?lartg`.

Input Parameters

a	REAL for srotg DOUBLE PRECISION for drotg COMPLEX for crotg DOUBLE COMPLEX for zrotg Provides the *x*-coordinate of the point p.
b	REAL for srotg DOUBLE PRECISION for drotg COMPLEX for crotg DOUBLE COMPLEX for zrotg Provides the *y*-coordinate of the point p.

Output Parameters

a	Contains the parameter *r* associated with the Givens rotation.
b	Contains the parameter *z* associated with the Givens rotation.
c	REAL for srotg DOUBLE PRECISION for drotg REAL for crotg DOUBLE PRECISION for zrotg Contains the parameter *c* associated with the Givens rotation.
s	REAL for srotg DOUBLE PRECISION for drotg COMPLEX for crotg DOUBLE COMPLEX for zrotg Contains the parameter *s* associated with the Givens rotation.

?rotm

Performs modified Givens rotation of points in the plane.

Syntax

```
call srotm(n, x, incx, y, incy, param)
call drotm(n, x, incx, y, incy, param)
call rotm(x, y, param)
```

Include Files

- mkl.fi, blas.f90

Description

Given two vectors *x* and *y*, each vector element of these vectors is replaced as follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = H \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

for *i*=1 to *n*, where *H* is a modified Givens transformation matrix whose values are stored in the `param(2)` through `param(5)` array. See discussion on the *param* argument.

Input Parameters

n
: `INTEGER`. Specifies the number of elements in vectors *x* and *y*.

x
: `REAL` for `srotm`
`DOUBLE PRECISION` for `drotm`
Array, size at least `(1 + (n -1)*abs(incx))`.

incx
: `INTEGER`. Specifies the increment for the elements of *x*.

y
: `REAL` for `srotm`
`DOUBLE PRECISION` for `drotm`
Array, size at least `(1 + (n -1)*abs(incy))`.

incy
: `INTEGER`. Specifies the increment for the elements of *y*.

param
: `REAL` for `srotm`
`DOUBLE PRECISION` for `drotm`
Array, size 5.
The elements of the *param* array are:
`param(1)` contains a switch, *flag*. `param(2-5)` contain h_{11}, h_{21}, h_{12}, and h_{22}, respectively, the components of the array *H*.
Depending on the values of *flag*, the components of *H* are set as follows:
flag = `-1.0`: $H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$
flag = `0.0`: $H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$
flag = `1.0`: $H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$
flag = `-2.0`: $H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$
In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of *flag* and are not required to be set in the *param* vector.

Output Parameters

x	Each element `x(i)` is replaced by h_{11}`*x(i) +` h_{12}`*y(i)`.
y	Each element `y(i)` is replaced by h_{21}`*x(i) +` h_{22}`*y(i)`.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `rotm` interface are the following:

x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.

?rotmg

Computes the parameters for a modified Givens rotation.

Syntax

```
call srotmg(d1, d2, x1, y1, param)
call drotmg(d1, d2, x1, y1, param)
call rotmg(d1, d2, x1, y1, param)
```

Include Files

- `mkl.fi, blas.f90`

Description

Given Cartesian coordinates (*x1*, *y1*) of an input vector, these routines compute the components of a modified Givens transformation matrix *H* that zeros the *y*-component of the resulting vector:

$$\begin{bmatrix} x1 \\ 0 \end{bmatrix} = H \begin{bmatrix} x1\sqrt{d1} \\ y1\sqrt{d2} \end{bmatrix}$$

Input Parameters

d1	`REAL` for `srotmg`
	`DOUBLE PRECISION` for `drotmg`
	Provides the scaling factor for the *x*-coordinate of the input vector.

d2	`REAL` for `srotmg`
	`DOUBLE PRECISION` for `drotmg`
	Provides the scaling factor for the *y*-coordinate of the input vector.

x1	`REAL` for `srotmg`
	`DOUBLE PRECISION` for `drotmg`
	Provides the *x*-coordinate of the input vector.

y1

REAL for srotmg

DOUBLE PRECISION for drotmg

Provides the *y*-coordinate of the input vector.

Output Parameters

d1

REAL for srotmg

DOUBLE PRECISION for drotmg

Provides the first diagonal element of the updated matrix.

d2

REAL for srotmg

DOUBLE PRECISION for drotmg

Provides the second diagonal element of the updated matrix.

x1

REAL for srotmg

DOUBLE PRECISION for drotmg

Provides the *x*-coordinate of the rotated vector before scaling.

param

REAL for srotmg

DOUBLE PRECISION for drotmg

Array, size 5.

The elements of the *param* array are:

param(1) contains a switch, *flag*. the other array elements *param*(2-5) contain the components of the array *H*: h_{11}, h_{21}, h_{12}, and h_{22}, respectively.

Depending on the values of *flag*, the components of *H* are set as follows:

flag = -1.0: $H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$

flag = 0.0: $H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$

flag = 1.0: $H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$

flag = -2.0: $H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of *flag* and are not required to be set in the *param* vector.

?scal

Computes the product of a vector by a scalar.

Syntax

```
call sscal(n, a, x, incx)
call dscal(n, a, x, incx)
```

```
call cscal(n, a, x, incx)
call zscal(n, a, x, incx)
call csscal(n, a, x, incx)
call zdscal(n, a, x, incx)
call scal(x, a)
```

Include Files

- mkl.fi, blas.f90

Description

The ?scal routines perform a vector operation defined as

```
x = a*x
```

where:

a is a scalar, *x* is an *n*-element vector.

Input Parameters

n	INTEGER. Specifies the number of elements in vector *x*.
a	REAL for sscal and csscal DOUBLE PRECISION for dscal and zdscal COMPLEX for cscal DOUBLE COMPLEX for zscal Specifies the scalar *a*.
x	REAL for sscal DOUBLE PRECISION for dscal COMPLEX for cscal and csscal DOUBLE COMPLEX for zscal and zdscal Array, size at least (1 + (*n* –1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

x	Updated vector x.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine scal interface are the following:

x	Holds the vector with the number of elements *n*.

?swap

Swaps a vector with another vector.

Syntax

```
call sswap(n, x, incx, y, incy)
call dswap(n, x, incx, y, incy)
call cswap(n, x, incx, y, incy)
call zswap(n, x, incx, y, incy)
call swap(x, y)
```

Include Files

- mkl.fi, blas.f90

Description

Given two vectors *x* and *y*, the ?swap routines return vectors *y* and *x* swapped, each replacing the other.

Input Parameters

n	INTEGER. Specifies the number of elements in vectors *x* and *y*.
x	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, size at least (1 + (*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, size at least (1 + (*n*-1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

x	Contains the resultant vector *x*, that is, the input vector *y*.
y	Contains the resultant vector *y*, that is, the input vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `swap` interface are the following:

x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.

i?amax

Finds the index of the element with maximum absolute value.

Syntax

```
index = isamax(n, x, incx)
index = idamax(n, x, incx)
index = icamax(n, x, incx)
index = izamax(n, x, incx)
index = iamax(x)
```

Include Files

- `mkl.fi, blas.f90`

Description

Given a vector *x*, the `i?amax` functions return the position of the vector element `x(i)` that has the largest absolute value for real flavors, or the largest sum `|Re(x[i])|+|Im(x[i])|` for complex flavors.

If either *n* or *incx* are not positive, the routine returns `0`.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

If the vector contains `NaN` values, then the routine returns the index of the first `NaN`.

Input Parameters

n	`INTEGER`. Specifies the number of elements in vector *x*.

x	`REAL` for `isamax`
	`DOUBLE PRECISION` for `idamax`
	`COMPLEX` for `icamax`
	`DOUBLE COMPLEX` for `izamax`
	Array, size at least `(1+(n-1)*abs(incx))`.

incx	`INTEGER`. Specifies the increment for the elements of *x*.

Output Parameters

index	`INTEGER`. Contains the position of vector element that has the largest absolute value such that `x(index)` has the largest absolute value.

BLAS 95 Interface Notes

Functions and routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the function `iamax` interface are the following:

x	Holds the vector with the number of elements *n*.

i?amin

Finds the index of the element with the smallest absolute value.

Syntax

```
index = isamin(n, x, incx)
index = idamin(n, x, incx)
index = icamin(n, x, incx)
index = izamin(n, x, incx)
index = iamin(x)
```

Include Files

- `mkl.fi, blas.f90`

Description

Given a vector *x*, the `i?amin` functions return the position of the vector element `x[i]` that has the smallest absolute value for real flavors, or the smallest sum `|Re(x[i])|+|Im(x[i])|` for complex flavors.

If either *n* or *incx* are not positive, the routine returns `0`.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

If the vector contains `NaN` values, then the routine returns the index of the first `NaN`.

Input Parameters

n	`INTEGER`. On entry, *n* specifies the number of elements in vector *x*.

x	`REAL` for `isamin`
	`DOUBLE PRECISION` for `idamin`
	`COMPLEX` for `icamin`
	`DOUBLE COMPLEX` for `izamin`
	Array, size at least `(1+(n-1)*abs(incx))`.

incx	`INTEGER`. Specifies the increment for the elements of *x*.

Output Parameters

index	`INTEGER`. Indicates the position of vector element with the smallest absolute value such that `x(index)` has the smallest absolute value.

BLAS 95 Interface Notes

Functions and routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the function `iamin` interface are the following:

x	Holds the vector with the number of elements *n*.

?cabs1

Computes absolute value of complex number.

Syntax

res = scabs1(*z*)

res = dcabs1(*z*)

res = cabs1(*z*)

Include Files

- mkl.fi, blas.f90

Description

The ?cabs1 is an auxiliary routine for a few BLAS Level 1 routines. This routine performs an operation defined as

```
res=|Re(z)|+|Im(z)|,
```

where *z* is a scalar, and *res* is a value containing the absolute value of a complex number *z*.

Input Parameters

z	COMPLEX scalar for scabs1. DOUBLE COMPLEX scalar for dcabs1.

Output Parameters

res	REAL for scabs1. DOUBLE PRECISION for dcabs1. Contains the absolute value of a complex number *z*.

BLAS Level 2 Routines

This section describes BLAS Level 2 routines, which perform matrix-vector operations. The following table lists the BLAS Level 2 routine groups and the data types associated with them.

BLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
?gbmv	s, d, c, z	Matrix-vector product using a general band matrix
?gemv	s, d, c, z	Matrix-vector product using a general matrix

Routine Groups	Data Types	Description
?ger	s, d	Rank-1 update of a general matrix
?gerc	c, z	Rank-1 update of a conjugated general matrix
?geru	c, z	Rank-1 update of a general matrix, unconjugated
?hbmv	c, z	Matrix-vector product using a Hermitian band matrix
?hemv	c, z	Matrix-vector product using a Hermitian matrix
?her	c, z	Rank-1 update of a Hermitian matrix
?her2	c, z	Rank-2 update of a Hermitian matrix
?hpmv	c, z	Matrix-vector product using a Hermitian packed matrix
?hpr	c, z	Rank-1 update of a Hermitian packed matrix
?hpr2	c, z	Rank-2 update of a Hermitian packed matrix
?sbmv	s, d	Matrix-vector product using symmetric band matrix
?spmv	s, d	Matrix-vector product using a symmetric packed matrix
?spr	s, d	Rank-1 update of a symmetric packed matrix
?spr2	s, d	Rank-2 update of a symmetric packed matrix
?symv	s, d	Matrix-vector product using a symmetric matrix
?syr	s, d	Rank-1 update of a symmetric matrix
?syr2	s, d	Rank-2 update of a symmetric matrix
?tbmv	s, d, c, z	Matrix-vector product using a triangular band matrix
?tbsv	s, d, c, z	Solution of a linear system of equations with a triangular band matrix
?tpmv	s, d, c, z	Matrix-vector product using a triangular packed matrix
?tpsv	s, d, c, z	Solution of a linear system of equations with a triangular packed matrix
?trmv	s, d, c, z	Matrix-vector product using a triangular matrix
?trsv	s, d, c, z	Solution of a linear system of equations with a triangular matrix

?gbmv

Computes a matrix-vector product with a general band matrix.

Syntax

```
call sgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call dgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call cgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call zgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
```

```
call gbmv(a, x, y [,kl] [,m] [,alpha] [,beta] [,trans])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gbmv routines perform a matrix-vector operation defined as

y := *alpha***A***x* + *beta***y*,

or

y := *alpha***A*'**x* + *beta***y*,

or

y := *alpha* *conjg(*A*')**x* + *beta***y*,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*n* band matrix, with *kl* sub-diagonals and *ku* super-diagonals.

Input Parameters

trans	CHARACTER*1. Specifies the operation: If *trans*= 'N' or 'n', then *y* := *alpha***A***x* + *beta***y* If *trans*= 'T' or 't', then *y* := *alpha***A*'**x* + *beta***y* If *trans*= 'C' or 'c', then *y* := *alpha* *conjg(*A*')**x* + *beta***y*
m	INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.
kl	INTEGER. Specifies the number of sub-diagonals of the matrix *A*. The value of *kl* must satisfy 0≤*kl*.
ku	INTEGER. Specifies the number of super-diagonals of the matrix *A*. The value of *ku* must satisfy 0≤*ku*.
alpha	REAL for sgbmv DOUBLE PRECISION for dgbmv COMPLEX for cgbmv DOUBLE COMPLEX for zgbmv Specifies the scalar *alpha*.
a	REAL for sgbmv DOUBLE PRECISION for dgbmv

COMPLEX for cgbmv

DOUBLE COMPLEX for zgbmv

Array, size (*lda*, *n*).

Before entry, the leading (*kl* + *ku* + 1) by *n* part of the array *a* must contain the matrix of coefficients. This matrix must be supplied column-by-column, with the leading diagonal of the matrix in row (*ku* + 1) of the array, the first super-diagonal starting at position 2 in row *ku*, the first sub-diagonal starting at position 1 in row (*ku* + 2), and so on. Elements in the array *a* that do not correspond to elements in the band matrix (such as the top left *ku* by *ku* triangle) are not referenced.

The following program segment transfers a band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
      do 20, j = 1, n
        k = ku + 1 - j
        do 10, i = max(1, j-ku), min(m, j+kl)
           a(k+i, j) = matrix(i,j)
10       continue
20    continue
```

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*kl* + *ku* + 1).

x — REAL for sgbmv

DOUBLE PRECISION for dgbmv

COMPLEX for cgbmv

DOUBLE COMPLEX for zgbmv

Array, size at least (1 + (*n* – 1)*abs(*incx*)) when *trans*= 'N' or 'n', and at least (1 + (*m* – 1)*abs(*incx*)) otherwise. Before entry, the array *x* must contain the vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. *incx* must not be zero.

beta — REAL for sgbmv

DOUBLE PRECISION for dgbmv

COMPLEX for cgbmv

DOUBLE COMPLEX for zgbmv

Specifies the scalar beta. When *beta* is equal to zero, then *y* need not be set on input.

y — REAL for sgbmv

DOUBLE PRECISION for dgbmv

COMPLEX for cgbmv

DOUBLE COMPLEX for zgbmv

	Array, size at least `(1 +(m - 1)*abs(incy))` when *trans*= `'N'` or `'n'` and at least `(1 +(n - 1)*abs(incy))` otherwise. Before entry, the incremented array *y* must contain the vector *y*.
incy	`INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Buffer holding the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `gbmv` interface are the following:

a	Holds the array *a* of size (*kl*+*ku*+*1*, *n*). Contains a banded matrix *m***n*with *kl* lower diagonal and *ku* upper diagonal.
x	Holds the vector with the number of elements *rx*, where *rx = n* if *trans = 'N'*, *rx = m* otherwise.
y	Holds the vector with the number of elements *ry*, where *ry = m* if *trans = 'N'*, *ry = n* otherwise.
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.
kl	If omitted, assumed *kl* = *ku*, that is, the number of lower diagonals equals the number of the upper diagonals.
ku	Restored as *ku* = *lda-kl-1*, where *lda* is the leading dimension of matrix *A*.
m	If omitted, assumed *m* = *n*, that is, a square matrix.
alpha	The default value is 1.
beta	The default value is 0.

?gemv

Computes a matrix-vector product using a general matrix.

Syntax

```
call sgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call dgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call cgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call zgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call scgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call dzgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call gemv(a, x, y [,alpha][,beta] [,trans])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gemv routines perform a matrix-vector operation defined as:

y := *alpha***A***x* + *beta***y*,

or

y := *alpha***A*'**x* + *beta***y*,

or

y := *alpha**conjg(*A*')**x* + *beta***y*,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*n* matrix.

Input Parameters

trans

CHARACTER*1 Specifies the operation:

if *trans*= 'N' or 'n', then *y* := *alpha***A***x* + *beta***y*;

if *trans*= 'T' or 't', then *y* := *alpha***A*'**x* + *beta***y*;

if *trans*= 'C' or 'c', then *y* := *alpha* *conjg(*A*')**x* + *beta***y*.

m

INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.

n

INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for sgemv

DOUBLE PRECISION for dgemv

COMPLEX for cgemv, scgemv

DOUBLE COMPLEX for zgemv, dzgemv

Specifies the scalar *alpha*.

a

REAL for sgemv, scgemv

DOUBLE PRECISION for dgemv, dzgemv

COMPLEX for cgemv

DOUBLE COMPLEX for zgemv

Array, size (*lda*, *n*).

Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

The value of *lda* must be at least `max(1, m)`.

x — REAL for sgemv

DOUBLE PRECISION for dgemv

COMPLEX for cgemv, scgemv

DOUBLE COMPLEX for zgemv, dzgemv

Array, size at least `(1+(n-1)*abs(incx))` when *trans*= 'N' or 'n' and at least `(1+(m - 1)*abs(incx))` otherwise. Before entry, the incremented array *x* must contain the vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

beta — REAL for sgemv

DOUBLE PRECISION for dgemv

COMPLEX for cgemv, scgemv

DOUBLE COMPLEX for zgemv, dzgemv

Specifies the scalar *beta*. When *beta* is set to zero, then *y* need not be set on input.

y — REAL for sgemv

DOUBLE PRECISION for dgemv

COMPLEX for cgemv, scgemv

DOUBLE COMPLEX for zgemv, dzgemv

Array, size at least `(1 +(m - 1)*abs(incy))` when *trans*= 'N' or 'n' and at least `(1 +(n - 1)*abs(incy))` otherwise. Before entry with non-zero *beta*, the incremented array *y* must contain the vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*.

The value of *incy* must not be zero.

Output Parameters

y — Updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine gemv interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

x — Holds the vector with the number of elements *rx* where *rx* = *n* if *trans* = 'N', *rx* = *m* otherwise.

y	Holds the vector with the number of elements *ry* where *ry* = *m* if *trans* = 'N', *ry* = *n* otherwise.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 0.

?ger

Performs a rank-1 update of a general matrix.

Syntax

```
call sger(m, n, alpha, x, incx, y, incy, a, lda)
call dger(m, n, alpha, x, incx, y, incy, a, lda)
call ger(a, x, y [,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?ger routines perform a matrix-vector operation defined as

```
A := alpha*x*y'+ A,
```

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

A is an *m*-by-*n* general matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.
alpha	REAL for sger DOUBLE PRECISION for dger Specifies the scalar *alpha*.
x	REAL for sger DOUBLE PRECISION for dger

	Array, size at least `(1 + (m - 1)*abs(incx))`. Before entry, the incremented array *x* must contain the *m*-element vector *x*.
incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y	`REAL` for `sger`
	`DOUBLE PRECISION` for `dger`
	Array, size at least `(1 + (n - 1)*abs(incy))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy	`INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

a	`REAL` for `sger`
	`DOUBLE PRECISION` for `dger`
	Array, size `(lda, n)`.
	Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.

lda	`INTEGER`. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, m)`.

Output Parameters

a	Overwritten by the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `ger` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
x	Holds the vector with the number of elements *m*.
y	Holds the vector with the number of elements *n*.
alpha	The default value is 1.

?gerc

Performs a rank-1 update (conjugated) of a general matrix.

Syntax

```
call cgerc(m, n, alpha, x, incx, y, incy, a, lda)
call zgerc(m, n, alpha, x, incx, y, incy, a, lda)
call gerc(a, x, y [,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gerc routines perform a matrix-vector operation defined as

```
A := alpha*x*conjg(y') + A,
```

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

A is an *m*-by-*n* matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.
alpha	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Specifies the scalar *alpha*.
x	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Array, size at least (1 + (*m* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *m*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Array, size at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.
a	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Array, size (*lda*, *n*). Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.

lda	INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).

Output Parameters

a	Overwritten by the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine gerc interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
x	Holds the vector with the number of elements *m*.
y	Holds the vector with the number of elements *n*.
alpha	The default value is 1.

?geru

Performs a rank-1 update (unconjugated) of a general matrix.

Syntax

```
call cgeru(m, n, alpha, x, incx, y, incy, a, lda)
call zgeru(m, n, alpha, x, incx, y, incy, a, lda)
call geru(a, x, y [,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?geru routines perform a matrix-vector operation defined as

```
A := alpha*x*y ' + A,
```

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

A is an *m*-by-*n* matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *A*.

The value of *m* must be at least zero.

n — INTEGER. Specifies the number of columns of the matrix *A*.

The value of *n* must be at least zero.

alpha — COMPLEX for cgeru

DOUBLE COMPLEX for zgeru

Specifies the scalar *alpha*.

x — COMPLEX for cgeru

DOUBLE COMPLEX for zgeru

Array, size at least (1 + (*m* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *m*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

y — COMPLEX for cgeru

DOUBLE COMPLEX for zgeru

Array, size at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*.

The value of *incy* must not be zero.

a — COMPLEX for cgeru

DOUBLE COMPLEX for zgeru

Array, size (*lda*, *n*).

Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

The value of *lda* must be at least max(1, *m*).

Output Parameters

a — Overwritten by the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine geru interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

x — Holds the vector with the number of elements *m*.

y — Holds the vector with the number of elements *n*.

alpha	The default value is 1.

?hbmv

Computes a matrix-vector product using a Hermitian band matrix.

Syntax

```
call chbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
call zhbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
call hbmv(a, x, y [,uplo][,alpha] [,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?hbmv routines perform a matrix-vector operation defined as *y := alpha*A*x + beta*y*,

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian band matrix, with *k* super-diagonals.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian band matrix *A* is used: If *uplo* = 'U' or 'u', then the upper triangular part of the matrix *A* is used. If *uplo* = 'L' or 'l', then the low triangular part of the matrix *A* is used.
n	INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.
k	INTEGER. For *uplo* = 'U' or 'u'Specifies the number of super-diagonals of the matrix *A*. For *uplo* = 'L' or 'l': Specifies the number of sub-diagonals of the matrix *A*. The value of *k* must satisfy $0 \le k$.
alpha	COMPLEX for chbmv DOUBLE COMPLEX for zhbmv Specifies the scalar *alpha*.
a	COMPLEX for chbmv DOUBLE COMPLEX for zhbmv Array, size (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the Hermitian matrix. The matrix must be supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the upper triangular part of a Hermitian band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
      do 20, j = 1, n
         m = k + 1 - j
         do 10, i = max( 1, j - k ), j
            a( m + i, j ) = matrix( i, j )
10       continue
20    continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the Hermitian matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the lower triangular part of a Hermitian band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
      do 20, j = 1, n
         m = 1 - j
         do 10, i = j, min( n, j + k )
            a( m + i, j ) = matrix( i, j )
10       continue
20    continue
```

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x — COMPLEX for chbmv

DOUBLE COMPLEX for zhbmv

Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

beta — COMPLEX for chbmv

DOUBLE COMPLEX for zhbmv

Specifies the scalar *beta*.

y — COMPLEX for chbmv

DOUBLE COMPLEX for zhbmv

	Array, size at least `(1 + (n - 1)*abs(incy))`. Before entry, the incremented array *y* must contain the vector *y*.
incy	`INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `hbmv` interface are the following:

a	Holds the array *a* of size (*k*+1,*n*).
x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
alpha	The default value is 1.
beta	The default value is 0.

?hemv

Computes a matrix-vector product using a Hermitian matrix.

Syntax

```
call chemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call zhemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call hemv(a, x, y [,uplo][,alpha] [,beta])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?hemv` routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is used.

If *uplo* = 'U' or 'u', then the upper triangular of the array *a* is used.

If *uplo* = 'L' or 'l', then the low triangular of the array *a* is used.

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Specifies the scalar *alpha*.

a

COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Array, size (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *a* is not referenced.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

x

COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

beta

COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Specifies the scalar *beta*. When *beta* is supplied as zero then *y* need not be set on input.

y

COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Array, size at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy

INTEGER. Specifies the increment for the elements of *y*.

	The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `hemv` interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
alpha	The default value is 1.
beta	The default value is 0.

?her

Performs a rank-1 update of a Hermitian matrix.

Syntax

```
call cher(uplo, n, alpha, x, incx, a, lda)
call zher(uplo, n, alpha, x, incx, a, lda)
call her(a, x [,uplo] [, alpha])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?her` routines perform a matrix-vector operation defined as

```
A := alpha*x*conjg(x') + A,
```

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

uplo	`CHARACTER*1`. Specifies whether the upper or lower triangular part of the array *a* is used. If *uplo* = `'U'` or `'u'`, then the upper triangular of the array *a* is used.

If *uplo* = 'L' or 'l', then the low triangular of the array *a* is used.

n
INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha
REAL for cher

DOUBLE PRECISION for zher

Specifies the scalar *alpha*.

x
COMPLEX for cher

DOUBLE COMPLEX for zher

Array, dimension at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx
INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

a
COMPLEX for cher

DOUBLE COMPLEX for zher

Array, size (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *a* is not referenced.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda
INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

Output Parameters

a
With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

If *alpha* is zero, matrix *A* is unchanged; otherwise, the imaginary parts of the diagonal elements are set to zero.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine her interface are the following:

a
Holds the matrix *A* of size (*n*,*n*).

x	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?her2

Performs a rank-2 update of a Hermitian matrix.

Syntax

```
call cher2(uplo, n, alpha, x, incx, y, incy, a, lda)
call zher2(uplo, n, alpha, x, incx, y, incy, a, lda)
call her2(a, x, y [,uplo][,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?her2 routines perform a matrix-vector operation defined as

```
A := alpha *x*conjg(y') + conjg(alpha)*y *conjg(x') + A,
```

where:

alpha is scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is used. If *uplo* = 'U' or 'u', then the upper triangular of the array *a* is used. If *uplo* = 'L' or 'l', then the low triangular of the array *a* is used.
n	INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.
alpha	COMPLEX for cher2 DOUBLE COMPLEX for zher2 Specifies the scalar *alpha*.
x	COMPLEX for cher2 DOUBLE COMPLEX for zher2 Array, size at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

y

COMPLEX for cher2

DOUBLE COMPLEX for zher2

Array, size at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy

INTEGER. Specifies the increment for the elements of *y*.

The value of *incy* must not be zero.

a

COMPLEX for cher2

DOUBLE COMPLEX for zher2

Array, size (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *a* is not referenced.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

Output Parameters

a

With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

If *alpha* is zero, matrix *A* is unchanged; otherwise, the imaginary parts of the diagonal elements are set to zero.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine her2 interface are the following:

a

Holds the matrix *A* of size (*n*,*n*).

x

Holds the vector with the number of elements *n*.

y

Holds the vector with the number of elements *n*.

uplo

Must be 'U' or 'L'. The default value is 'U'.

alpha

The default value is 1.

?hpmv

Computes a matrix-vector product using a Hermitian packed matrix.

Syntax

```
call chpmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
call zhpmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
call hpmv(ap, x, y [,uplo][,alpha] [,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?hpmv routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*.

If *uplo* = 'U' or 'u', then the upper triangular part of the matrix *A* is supplied in the packed array *ap* .

If *uplo* = 'L' or 'l', then the low triangular part of the matrix *A* is supplied in the packed array *ap* .

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

COMPLEX for chpmv

DOUBLE COMPLEX for zhpmv

Specifies the scalar *alpha*.

ap

COMPLEX for chpmv

DOUBLE COMPLEX for zhpmv

Array, size at least ((*n**(*n* + 1))/2).

Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.

	The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

x	`COMPLEX` for `chpmv`
	`DOUBLE PRECISION COMPLEX` for `zhpmv`
	Array, size at least `(1 +(n - 1)*abs(incx))`. Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta	`COMPLEX` for `chpmv`
	`DOUBLE COMPLEX` for `zhpmv`
	Specifies the scalar *beta*.
	When *beta* is equal to zero then *y* need not be set on input.

y	`COMPLEX` for `chpmv`
	`DOUBLE COMPLEX` for `zhpmv`
	Array, size at least `(1 + (n - 1)*abs(incy))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy	`INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `hpmv` interface are the following:

ap	Holds the array *ap* of size (*n**(n+1)/2*).
x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
alpha	The default value is 1.
beta	The default value is 0.

?hpr

Performs a rank-1 update of a Hermitian packed matrix.

Syntax

```
call chpr(uplo, n, alpha, x, incx, ap)
```

```
call zhpr(uplo, n, alpha, x, incx, ap)
call hpr(ap, x [,uplo] [, alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?hpr routines perform a matrix-vector operation defined as

```
A := alpha*x*conjg(x') + A,
```

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*.

If *uplo* = 'U' or 'u', the upper triangular part of the matrix *A* is supplied in the packed array *ap* .

If *uplo* = 'L' or 'l', the low triangular part of the matrix *A* is supplied in the packed array *ap* .

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for chpr

DOUBLE PRECISION for zhpr

Specifies the scalar *alpha*.

x

COMPLEX for chpr

DOUBLE COMPLEX for zhpr

Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*. *incx* must not be zero.

ap

COMPLEX for chpr

DOUBLE COMPLEX for zhpr

Array, size at least ((*n**(*n* + 1))/2).

Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on.

Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap

With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

If *alpha* is zero, matrix *A* is unchanged; otherwise, the imaginary parts of the diagonal elements are set to zero.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine hpr interface are the following:

ap — Holds the array *ap* of size (*n**(*n*+1)/2).

x — Holds the vector with the number of elements *n*.

uplo — Must be 'U' or 'L'. The default value is 'U'.

alpha — The default value is 1.

?hpr2

Performs a rank-2 update of a Hermitian packed matrix.

Syntax

```
call chpr2(uplo, n, alpha, x, incx, y, incy, ap)
call zhpr2(uplo, n, alpha, x, incx, y, incy, ap)
call hpr2(ap, x, y [,uplo][,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?hpr2 routines perform a matrix-vector operation defined as

```
A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A,
```

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*.

If *uplo* = 'U' or 'u', then the upper triangular part of the matrix *A* is supplied in the packed array *ap* .

If *uplo* = 'L' or 'l', then the low triangular part of the matrix *A* is supplied in the packed array *ap* .

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

COMPLEX for chpr2

DOUBLE COMPLEX for zhpr2

Specifies the scalar *alpha*.

x

COMPLEX for chpr2

DOUBLE COMPLEX for zhpr2

Array, dimension at least (1 +(*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

y

COMPLEX for chpr2

DOUBLE COMPLEX for zhpr2

Array, size at least (1 +(*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy

INTEGER. Specifies the increment for the elements of *y*.

The value of *incy* must not be zero.

ap

COMPLEX for chpr2

DOUBLE COMPLEX for zhpr2

Array, size at least ((*n**(*n* + 1))/2).

Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on.

Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap	With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix. If *alpha* is zero, matrix *A* is unchanged; otherwise, the imaginary parts of the diagonal elements need are set to zero.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine hpr2 interface are the following:

ap	Holds the array *ap* of size (*n*(n+1)/2*).
x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?sbmv

Computes a matrix-vector product with a symmetric band matrix.

Syntax

```
call ssbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
call dsbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
call sbmv(a, x, y [,uplo][,alpha] [,beta])
```

Include Files

- mkl.fi,blas.f90

Description

The ?sbmv routines perform a matrix-vector operation defined as

*y := alpha*A*x + beta*y*,

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric band matrix, with *k* super-diagonals.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix *A* is used:

if *uplo* = 'U' or 'u' - upper triangular part;

if *uplo* = 'L' or 'l' - low triangular part.

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

k

INTEGER. Specifies the number of super-diagonals of the matrix *A*.

The value of *k* must satisfy $0 \le k$.

alpha

REAL for ssbmv

DOUBLE PRECISION for dsbmv

Specifies the scalar *alpha*.

a

REAL for ssbmv

DOUBLE PRECISION for dsbmv

Array, size (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the upper triangular part of a symmetric band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
      do 20, j = 1, n
         m = k + 1 - j
         do 10, i = max( 1, j - k ), j
            a( m + i, j ) = matrix( i, j )
10       continue
20    continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the lower triangular part of a symmetric band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
      do 20, j = 1, n
         m = 1 - j
         do 10, i = j, min( n, j + k )
            a( m + i, j ) = matrix( i, j )
10       continue
20    continue
```

lda	INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).
x	REAL for ssbmv DOUBLE PRECISION for dsbmv Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
beta	REAL for ssbmv DOUBLE PRECISION for dsbmv Specifies the scalar *beta*.
y	REAL for ssbmv DOUBLE PRECISION for dsbmv Array, size at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine sbmv interface are the following:

a	Holds the array *a* of size (*k*+1,*n*).
x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.
beta	The default value is 0.

?spmv

Computes a matrix-vector product with a symmetric packed matrix.

Syntax

```
call sspmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
```

```
call dspmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
call spmv(ap, x, y [,uplo][,alpha] [,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?spmv routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*. If *uplo* = 'U' or 'u', then the upper triangular part of the matrix *A* is supplied in the packed array *ap* . If *uplo* = 'L' or 'l', then the low triangular part of the matrix *A* is supplied in the packed array *ap* .
n	INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.
alpha	REAL for sspmv DOUBLE PRECISION for dspmv Specifies the scalar *alpha*.
ap	REAL for sspmv DOUBLE PRECISION for dspmv Array, size at least ((n*(n + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1,2) and *a*(2, 2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(2,1) and *a*(3,1) respectively, and so on.
x	REAL for sspmv DOUBLE PRECISION for dspmv

	Array, size at least `(1 + (n - 1)*abs(incx))`. Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta	`REAL` for `sspmv`
	`DOUBLE PRECISION` for `dspmv`
	Specifies the scalar *beta*.
	When *beta* is supplied as zero, then *y* need not be set on input.

y	`REAL` for `sspmv`
	`DOUBLE PRECISION` for `dspmv`
	Array, size at least `(1 + (n - 1)*abs(incy))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy	`INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `spmv` interface are the following:

ap	Holds the array *ap* of size (*n**(n+1)/2*).
x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
alpha	The default value is 1.
beta	The default value is 0.

?spr

Performs a rank-1 update of a symmetric packed matrix.

Syntax

```
call sspr(uplo, n, alpha, x, incx, ap)
call dspr(uplo, n, alpha, x, incx, ap)
call spr(ap, x [,uplo] [, alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?spr routines perform a matrix-vector operation defined as

```
a:= alpha*x*x'+ A,
```

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*.

If *uplo* = 'U' or 'u', then the upper triangular part of the matrix *A* is supplied in the packed array *ap* .

If *uplo* = 'L' or 'l', then the low triangular part of the matrix *A* is supplied in the packed array *ap* .

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for sspr

DOUBLE PRECISION for dspr

Specifies the scalar *alpha*.

x

REAL for sspr

DOUBLE PRECISION for dspr

Array, size at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

ap

REAL for sspr

DOUBLE PRECISION for dspr

Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on.

Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.

Output Parameters

ap	With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine spr interface are the following:

ap	Holds the array *ap* of size (*n*(n+1)/2*).
x	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?spr2

Computes a rank-2 update of a symmetric packed matrix.

Syntax

```
call sspr2(uplo, n, alpha, x, incx, y, incy, ap)
call dspr2(uplo, n, alpha, x, incx, y, incy, ap)
call spr2(ap, x, y [,uplo][,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?spr2 routines perform a matrix-vector operation defined as

```
A:= alpha*x*y'+ alpha*y*x' + A,
```

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*. If *uplo* = 'U' or 'u', then the upper triangular part of the matrix *A* is supplied in the packed array *ap* .

If *uplo* = 'L' or 'l', then the low triangular part of the matrix *A* is supplied in the packed array *ap* .

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for sspr2

DOUBLE PRECISION for dspr2

Specifies the scalar *alpha*.

x

REAL for sspr2

DOUBLE PRECISION for dspr2

Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

y

REAL for sspr2

DOUBLE PRECISION for dspr2

Array, size at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy

INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

ap

REAL for sspr2

DOUBLE PRECISION for dspr2

Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on.

Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains $A_{1,1}$, *ap*(2) and *ap*(3) contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.

Output Parameters

ap

With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine spr2 interface are the following:

ap	Holds the array *ap* of size (*n**(*n*+1)/2).
x	Holds the vector with the number of elements *n*.
y	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?symv

Computes a matrix-vector product for a symmetric matrix.

Syntax

```
call ssymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call dsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call symv(a, x, y [,uplo][,alpha] [,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?symv routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is used. If *uplo* = 'U' or 'u', then the upper triangular part of the array *a* is used. If *uplo* = 'L' or 'l', then the low triangular part of the array *a* is used.
n	INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.
alpha	REAL for ssymv DOUBLE PRECISION for dsymv Specifies the scalar *alpha*.
a	REAL for ssymv DOUBLE PRECISION for dsymv Array, size (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix *A* and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix *A* and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

x — REAL for ssymv

DOUBLE PRECISION for dsymv

Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

beta — REAL for ssymv

DOUBLE PRECISION for dsymv

Specifies the scalar *beta*.

When *beta* is supplied as zero, then *y* need not be set on input.

y — REAL for ssymv

DOUBLE PRECISION for dsymv

Array, size at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*.

The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine symv interface are the following:

a — Holds the matrix *A* of size (*n*,*n*).

x — Holds the vector with the number of elements *n*.

y — Holds the vector with the number of elements *n*.

uplo — Must be 'U' or 'L'. The default value is 'U'.

alpha — The default value is 1.

beta — The default value is 0.

?syr

Performs a rank-1 update of a symmetric matrix.

Syntax

```
call ssyr(uplo, n, alpha, x, incx, a, lda)
call dsyr(uplo, n, alpha, x, incx, a, lda)
call syr(a, x [,uplo] [, alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?syr routines perform a matrix-vector operation defined as

*A := alpha*x*x' + A*,

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is used.

If *uplo* = 'U' or 'u', then the upper triangular part of the array *a* is used.

If *uplo* = 'L' or 'l', then the low triangular part of the array *a* is used.

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for ssyr

DOUBLE PRECISION for dsyr

Specifies the scalar *alpha*.

x

REAL for ssyr

DOUBLE PRECISION for dsyr

Array, size at least (1 + (*n*-1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

a

REAL for ssyr

DOUBLE PRECISION for dsyr

Array, size (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix *A* and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix *A* and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

Output Parameters

a — With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine syr interface are the following:

a — Holds the matrix *A* of size (*n*,*n*).

x — Holds the vector with the number of elements *n*.

uplo — Must be 'U' or 'L'. The default value is 'U'.

alpha — The default value is 1.

?syr2

Performs a rank-2 update of a symmetric matrix.

Syntax

```
call ssyr2(uplo, n, alpha, x, incx, y, incy, a, lda)
call dsyr2(uplo, n, alpha, x, incx, y, incy, a, lda)
call syr2(a, x, y [,uplo][,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?syr2 routines perform a matrix-vector operation defined as

```
A := alpha*x*y'+ alpha*y*x' + A,
```

where:

alpha is scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

uplo
: CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is used.

  If *uplo* = 'U' or 'u', then the upper triangular part of the array *a* is used.

  If *uplo* = 'L' or 'l', then the low triangular part of the array *a* is used.

n
: INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha
: REAL for ssyr2

  DOUBLE PRECISION for dsyr2

  Specifies the scalar *alpha*.

x
: REAL for ssyr2

  DOUBLE PRECISION for dsyr2

  Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx
: INTEGER. Specifies the increment for the elements of *x*.

  The value of *incx* must not be zero.

y
: REAL for ssyr2

  DOUBLE PRECISION for dsyr2

  Array, size at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy
: INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

a
: REAL for ssyr2

  DOUBLE PRECISION for dsyr2

  Array, size (*lda*, *n*).

  Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced.

  Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda
: INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

Output Parameters

a
: With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine syr2 interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
x	Holds the vector *x* of length *n*.
y	Holds the vector *y* of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?tbmv

Computes a matrix-vector product using a triangular band matrix.

Syntax

```
call stbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call dtbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call ctbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call ztbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call tbmv(a, x [,uplo] [, trans] [,diag])
```

Include Files

- mkl.fi, blas.f90

Description

The ?tbmv routines perform one of the matrix-vector operations defined as

x := *A***x*, or *x* := *A*'**x*, or *x* := conjg(*A*')**x*,

where:

x is an *n*-element vector,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular band matrix, with (*k* +1) diagonals.

Input Parameters

uplo	CHARACTER*1. Specifies whether the matrix *A* is an upper or lower triangular matrix: *uplo* = 'U' or 'u' if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans

CHARACTER*1. Specifies the operation:

if *trans*= 'N' or 'n', then *x* := *A***x*;

if *trans*= 'T' or 't', then *x* := *A*'**x*;

if *trans*= 'C' or 'c', then *x* := conjg(*A*')**x*.

diag

CHARACTER*1. Specifies whether the matrix *A* is unit triangular:

if *uplo* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

k

INTEGER. On entry with *uplo* = 'U' or 'u' specifies the number of super-diagonals of the matrix *A*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *a*.

The value of *k* must satisfy $0 \le k$.

a

REAL for stbmv

DOUBLE PRECISION for dtbmv

COMPLEX for ctbmv

DOUBLE COMPLEX for ztbmv

Array, size (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced. The following program segment transfers an upper triangular band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
      do 20, j = 1, n
         m = k + 1 - j
         do 10, i = max( 1, j - k ), j
            a( m + i, j ) = matrix( i, j )
10       continue
20    continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced. The following program segment transfers a lower triangular band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
      do 20, j = 1, n
         m = 1 - j
         do 10, i = j, min( n, j + k )
```

```
            a( m + i, j ) = matrix( i, j )
10        continue
20     continue
```

Note that when *uplo* = 'U' or 'u', the elements of the array *a* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x — REAL for stbmv

DOUBLE PRECISION for dtbmv

COMPLEX for ctbmv

DOUBLE COMPLEX for ztbmv

Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

Output Parameters

x — Overwritten with the transformed vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine tbmv interface are the following:

a — Holds the array *a* of size (*k*+1,*n*).

x — Holds the vector with the number of elements *n*.

uplo — Must be 'U' or 'L'. The default value is 'U'.

trans — Must be 'N', 'C', or 'T'.

The default value is 'N'.

diag — Must be 'N' or 'U'. The default value is 'N'.

?tbsv

Solves a system of linear equations whose coefficients are in a triangular band matrix.

Syntax

```
call stbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call dtbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call ctbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call ztbsv(uplo, trans, diag, n, k, a, lda, x, incx)
```

```
call tbsv(a, x [,uplo] [, trans] [,diag])
```

Include Files

- mkl.fi, blas.f90

Description

The ?tbsv routines solve one of the following systems of equations:

*A*x* = *b*, or *A*'*x* = *b*, or conjg(*A*')*x* = *b*,

where:

b and *x* are *n*-element vectors,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular band matrix, with (*k* + 1) diagonals.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

uplo	CHARACTER*1. Specifies whether the matrix *A* is an upper or lower triangular matrix: if *uplo* = 'U' or 'u' the matrix is upper triangular; if *uplo* = 'L' or 'l', the matrix is low triangular.
trans	CHARACTER*1. Specifies the system of equations: if *trans*= 'N' or 'n', then *A*x* = *b*; if *trans*= 'T' or 't', then *A*'*x* = *b*; if *trans*= 'C' or 'c', then conjg(*A*')*x* = *b*.
diag	CHARACTER*1. Specifies whether the matrix *A* is unit triangular: if *diag* = 'U' or 'u' then the matrix is unit triangular; if *diag* = 'N' or 'n', then the matrix is not unit triangular.
n	INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.
k	INTEGER. On entry with *uplo* = 'U' or 'u', *k* specifies the number of super-diagonals of the matrix *A*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *A*. The value of *k* must satisfy 0≤*k*.
a	REAL for stbsv DOUBLE PRECISION for dtbsv COMPLEX for ctbsv DOUBLE COMPLEX for ztbsv Array, size (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers an upper triangular band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
      do 20, j = 1, n
         m = k + 1 - j
         do 10, i = max( 1, j - k ), j
            a( m + i, j ) = matrix( i, j )
10       continue
20    continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers a lower triangular band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
      do 20, j = 1, n
         m = 1 - j
         do 10, i = j, min( n, j + k )
            a( m + i, j ) = matrix( i, j )
10       continue
20    continue
```

When *diag* = 'U' or 'u', the elements of the array *a* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x — REAL for stbsv

DOUBLE PRECISION for dtbsv

COMPLEX for ctbsv

DOUBLE COMPLEX for ztbsv

Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.

incx — INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

Output Parameters

x — Overwritten with the solution vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `tbsv` interface are the following:

a	Holds the array *a* of size (*k*+1,*n*).
x	Holds the vector with the number of elements *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.
diag	Must be `'N'` or `'U'`. The default value is `'N'`.

?tpmv

Computes a matrix-vector product using a triangular packed matrix.

Syntax

```
call stpmv(uplo, trans, diag, n, ap, x, incx)
call dtpmv(uplo, trans, diag, n, ap, x, incx)
call ctpmv(uplo, trans, diag, n, ap, x, incx)
call ztpmv(uplo, trans, diag, n, ap, x, incx)
call tpmv(ap, x [,uplo] [, trans] [,diag])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?tpmv` routines perform one of the matrix-vector operations defined as

`x := A*x`, or `x := A'*x`, or `x := conjg(A')*x`,

where:

x is an *n*-element vector,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

Input Parameters

uplo	`CHARACTER*1`. Specifies whether the matrix *A* is upper or lower triangular: *uplo* = `'U'` or `'u'` if *uplo* = `'L'` or `'l'`, then the matrix is low triangular.
trans	`CHARACTER*1`. Specifies the operation: if *trans*= `'N'` or `'n'`, then `x := A*x`;

if *trans*= 'T' or 't', then *x* := *A*'**x*;

if *trans*= 'C' or 'c', then *x* := conjg(*A*')**x*.

diag CHARACTER*1. Specifies whether the matrix *A* is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

ap REAL for stpmv

DOUBLE PRECISION for dtpmv

COMPLEX for ctpmv

DOUBLE COMPLEX for ztpmv

Array, size at least ((*n**(*n* + 1))/2).

Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1,2) and *a*(2,2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular matrix packed sequentially, column-by-column, so that*ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(2,1) and *a*(3,1) respectively, and so on. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced, but are assumed to be unity.

x REAL for stpmv

DOUBLE PRECISION for dtpmv

COMPLEX for ctpmv

DOUBLE COMPLEX for ztpmv

Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

Output Parameters

x Overwritten with the transformed vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine tpmv interface are the following:

ap Holds the array *ap* of size (*n*(n+1)/2*).

x Holds the vector with the number of elements *n*.

uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

?tpsv

Solves a system of linear equations whose coefficients are in a triangular packed matrix.

Syntax

```
call stpsv(uplo, trans, diag, n, ap, x, incx)
call dtpsv(uplo, trans, diag, n, ap, x, incx)
call ctpsv(uplo, trans, diag, n, ap, x, incx)
call ztpsv(uplo, trans, diag, n, ap, x, incx)
call tpsv(ap, x [,uplo] [, trans] [,diag])
```

Include Files

- mkl.fi, blas.f90

Description

The ?tpsv routines solve one of the following systems of equations

*A*x = b*, or *A'*x = b*, or conjg(*A'*)*x = b*,

where:

b and *x* are *n*-element vectors,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

This routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

uplo	CHARACTER*1. Specifies whether the matrix *A* is upper or lower triangular: *uplo* = 'U' or 'u' if *uplo* = 'L' or 'l', then the matrix is low triangular.
trans	CHARACTER*1. Specifies the system of equations: if *trans*= 'N' or 'n', then *A*x = b*; if *trans*= 'T' or 't', then *A'*x = b*; if *trans*= 'C' or 'c', then conjg(*A'*)*x = b*.
diag	CHARACTER*1. Specifies whether the matrix *A* is unit triangular: if *diag* = 'U' or 'u' then the matrix is unit triangular;

	if *diag* = 'N' or 'n', then the matrix is not unit triangular.
n	INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.
ap	REAL for stpsv DOUBLE PRECISION for dtpsv COMPLEX for ctpsv DOUBLE COMPLEX for ztpsv Array, size at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced, but are assumed to be unity.
x	REAL for stpsv DOUBLE PRECISION for dtpsv COMPLEX for ctpsv DOUBLE COMPLEX for ztpsv Array, size at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x	Overwritten with the solution vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine tpsv interface are the following:

ap	Holds the array *ap* of size (*n*(n+1)/2*).
x	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'.

The default value is 'N'.

diag Must be 'N' or 'U'. The default value is 'N'.

?trmv

Computes a matrix-vector product using a triangular matrix.

Syntax

```
call strmv(uplo, trans, diag, n, a, lda, x, incx)
call dtrmv(uplo, trans, diag, n, a, lda, x, incx)
call ctrmv(uplo, trans, diag, n, a, lda, x, incx)
call ztrmv(uplo, trans, diag, n, a, lda, x, incx)
call trmv(a, x [,uplo] [, trans] [,diag])
```

Include Files

- mkl.fi, blas.f90

Description

The ?trmv routines perform one of the following matrix-vector operations defined as

x := *A***x*, or *x* := A'**x*, or *x* := conjg(*A*')**x*,

where:

x is an *n*-element vector,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix.

Input Parameters

uplo CHARACTER*1. Specifies whether the matrix *A* is upper or lower triangular:

uplo = 'U' or 'u'

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans CHARACTER*1. Specifies the operation:

if *trans*= 'N' or 'n', then *x* := *A***x*;

if *trans*= 'T' or 't', then *x* := *A*'**x*;

if *trans*= 'C' or 'c', then *x* := conjg(*A*')**x*.

diag CHARACTER*1. Specifies whether the matrix *A* is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

a REAL for strmv

DOUBLE PRECISION for dtrmv

	COMPLEX for ctrmv DOUBLE COMPLEX for ztrmv Array, size (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.
lda	INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).
x	REAL for strmv DOUBLE PRECISION for dtrmv COMPLEX for ctrmv DOUBLE COMPLEX for ztrmv Array, size at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x	Overwritten with the transformed vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine trmv interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
x	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

?trsv

Solves a system of linear equations whose coefficients are in a triangular matrix.

Syntax

```
call strsv(uplo, trans, diag, n, a, lda, x, incx)
call dtrsv(uplo, trans, diag, n, a, lda, x, incx)
call ctrsv(uplo, trans, diag, n, a, lda, x, incx)
call ztrsv(uplo, trans, diag, n, a, lda, x, incx)
call trsv(a, x [,uplo] [, trans] [,diag])
```

Include Files

- mkl.fi, blas.f90

Description

The ?trsv routines solve one of the systems of equations:

*A*x = b*, or *A'*x = b*, or conjg(*A'*)*x = b*,

where:

b and *x* are *n*-element vectors,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

uplo

CHARACTER*1. Specifies whether the matrix *A* is upper or lower triangular:

uplo = 'U' or 'u'

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans

CHARACTER*1. Specifies the systems of equations:

if *trans*= 'N' or 'n', then *A*x = b*;

if *trans*= 'T' or 't', then *A'*x = b*;

if *trans*= 'C' or 'c', then oconj(*A'*)*x = b*.

diag

CHARACTER*1. Specifies whether the matrix *A* is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

a

REAL for strsv

DOUBLE PRECISION for dtrsv

COMPLEX for ctrsv

DOUBLE COMPLEX for ztrsv

	Array, size `(lda, n)` . Before entry with `uplo = 'U'` or `'u'`, the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with `uplo = 'L'` or `'l'`, the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced. When `diag = 'U'` or `'u'`, the diagonal elements of *a* are not referenced either, but are assumed to be unity.
lda	`INTEGER`. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, n)`.

x	`REAL` for `strsv`
	`DOUBLE PRECISION` for `dtrsv`
	`COMPLEX` for `ctrsv`
	`DOUBLE COMPLEX` for `ztrsv`
	Array, size at least `(1 + (n – 1)*abs(incx))`. Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.

incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x	Overwritten with the solution vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `trsv` interface are the following:

a	Holds the matrix *a* of size (*n*,*n*).
x	Holds the vector with the number of elements *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.
diag	Must be `'N'` or `'U'`. The default value is `'N'`.

BLAS Level 3 Routines

BLAS Level 3 routines perform matrix-matrix operations. The following table lists the BLAS Level 3 routine groups and the data types associated with them.

BLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
`?gemm`	s, d, c, z	Computes a matrix-matrix product with general matrices.

Routine Group	Data Types	Description
?hemm	c, z	Computes a matrix-matrix product where one input matrix is Hermitian.
?herk	c, z	Performs a Hermitian rank-k update.
?her2k	c, z	Performs a Hermitian rank-2k update.
?symm	s, d, c, z	Computes a matrix-matrix product where one input matrix is symmetric.
?syrk	s, d, c, z	Performs a symmetric rank-k update.
?syr2k	s, d, c, z	Performs a symmetric rank-2k update.
?trmm	s, d, c, z	Computes a matrix-matrix product where one input matrix is triangular.
?trsm	s, d, c, z	Solves a triangular matrix equation.

Symmetric Multiprocessing Version of Intel® MKL

Many applications spend considerable time executing BLAS routines. This time can be scaled by the number of processors available on the system through using the symmetric multiprocessing (SMP) feature built into the Intel® oneMKL. The performance enhancements based on the parallel use of the processors are available without any programming effort on your part.

To enhance performance, the library uses the following methods:

- The BLAS functions are blocked where possible to restructure the code in a way that increases the localization of data reference, enhances cache memory use, and reduces the dependency on the memory bus.
- The code is distributed across the processors to maximize parallelism.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

?gemm

Computes a matrix-matrix product with general matrices.

Syntax

```
call sgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call cgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call scgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dzgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call gemm(a, b, c [,transa][,transb] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gemm routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, with general matrices. The operation is defined as

```
C := alpha*op(A)*op(B) + beta*C
```

where:

op(*X*) is one of op(*X*) = *X*, or op(*X*) = X^T, or op(*X*) = X^H,

alpha and *beta* are scalars,

A, *B* and *C* are matrices:

op(*A*) is an *m*-by-*k* matrix,

op(*B*) is a *k*-by-*n* matrix,

C is an *m*-by-*n* matrix.

See also:

- ?gemm3m, BLAS-like extension routines, that use matrix multiplication for similar matrix-matrix operations

Input Parameters

transa

CHARACTER*1. Specifies the form of op(*A*) used in the matrix multiplication:

- if *transa* = 'N' or 'n', then op(*A*) = *A*;
- if *transa* = 'T' or 't', then op(*A*) = A^T;
- if *transa* = 'C' or 'c' , then op(*A*) = A^H.

transb

CHARACTER*1. Specifies the form of op(*B*) used in the matrix multiplication:

- if *transb* = 'N' or 'n', then op(*B*) = *B*;
- if *transb* = 'T' or 't', then op(*B*) = B^T;
- if *transb* = 'C' or 'c' , then op(*B*) = B^H.

m

INTEGER. Specifies the number of rows of the matrix op(*A*) and of the matrix *C*. The value of *m* must be at least zero.

n

INTEGER. Specifies the number of columns of the matrix op(*B*) and the number of columns of the matrix *C*. The value of *n* must be at least zero.

k

INTEGER. Specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*). The value of *k* must be at least zero.

alpha

REAL for sgemm

DOUBLE PRECISION for dgemm

COMPLEX for cgemm, scgemm

DOUBLE COMPLEX for zgemm, dzgemm

Specifies the scalar *alpha*.

a

REAL for sgemm, scgemm

DOUBLE PRECISION for dgemm, dzgemm

COMPLEX for cgemm

DOUBLE COMPLEX for zgemm

Array, size *lda* by *ka*, where *ka* is *k* when *transa* = 'N' or 'n', and is *m* otherwise. Before entry with *transa* = 'N' or 'n', the leading *m*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*m* part of the array *a* must contain the matrix *A*.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

When *transa* = 'N' or 'n', then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1, *k*).

b

REAL for sgemm

DOUBLE PRECISION for dgemm

COMPLEX for cgemm, scgemm

DOUBLE COMPLEX for zgemm, dzgemm

Array, size *ldb* by *kb*, where *kb* is *n* when *transa* = 'N' or 'n', and is *k* otherwise. Before entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *n*-by-*k* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

When *transb* = 'N' or 'n' , then *ldb* must be at least max(1, *k*), otherwise *ldb* must be at least max(1, *n*).

beta

REAL for sgemm

DOUBLE PRECISION for dgemm

COMPLEX for cgemm, scgemm

DOUBLE COMPLEX for zgemm, dzgemm

Specifies the scalar *beta*. When *beta* is equal to zero, then *c* need not be set on input.

c

REAL for sgemm

DOUBLE PRECISION for dgemm

COMPLEX for cgemm, scgemm

DOUBLE COMPLEX for zgemm, dzgemm

Array, size *ldc* by *n*. Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is equal to zero, in which case *c* need not be set on entry.

ldc

INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

The value of *ldc* must be at least max(1, *m*).

Output Parameters

c Overwritten by the *m*-by-*n* matrix (*alpha**op(*A*)*op(*B*) + *beta***C*).

Example

For examples of routine usage, see these code examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- sgemm: examples\blas\source\sgemmx.f
- dgemm: examples\blas\source\dgemmx.f
- cgemm: examples\blas\source\cgemmx.f
- zgemm: examples\blas\source\zgemmx.f

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine gemm interface are the following:

a Holds the matrix *A* of size (*ma*,*ka*) where

ka = *k* if *transa*='N',

ka = *m* otherwise,

ma = *m* if *transa*='N',

ma = *k* otherwise.

b Holds the matrix *B* of size (*mb*,*kb*) where

kb = *n* if *transb* = 'N',

kb = *k* otherwise,

mb = *k* if *transb* = 'N',

mb = *n* otherwise.

c Holds the matrix *C* of size (*m*,*n*).

transa Must be 'N', 'C', or 'T'.

The default value is 'N'.

transb Must be 'N', 'C', or 'T'.

The default value is 'N'.

alpha The default value is 1.

beta The default value is 0.

?hemm

Computes a matrix-matrix product where one input matrix is Hermitian.

Syntax

```
call chemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call zhemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call hemm(a, b, c [,side][,uplo] [,alpha][,beta])
```

Include Files

- mkl.fi,blas.f90

Description

The ?hemm routines compute a scalar-matrix-matrix product using a Hermitian matrix *A* and a general matrix *B* and add the result to a scalar-matrix product using a general matrix *C*. The operation is defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*B*A + beta*C
```

where:

alpha and *beta* are scalars,

A is a Hermitian matrix,

B and *C* are *m*-by-*n* matrices.

Input Parameters

side

CHARACTER*1. Specifies whether the Hermitian matrix *A* appears on the left or right in the operation as follows:

if *side* = 'L' or 'l' , then *C* := *alpha*A*B + beta*C*;

if *side* = 'R' or 'r' , then *C* := *alpha*B*A + beta*C*.

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the Hermitian matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangular part of the Hermitian matrix *A* is used.

m

INTEGER. Specifies the number of rows of the matrix *C*.

The value of *m* must be at least zero.

n

INTEGER. Specifies the number of columns of the matrix *C*.

The value of *n* must be at least zero.

alpha

COMPLEX for chemm

DOUBLE COMPLEX for zhemm

Specifies the scalar *alpha*.

a

COMPLEX for chemm

DOUBLE COMPLEX for zhemm

Array, size (*lda*, *ka*), where *ka* is *m* when *side* = 'L' or 'l' and is *n* otherwise. Before entry with *side* = 'L' or 'l', the *m*-by-*m* part of the array *a* must contain the Hermitian matrix, such that when *uplo* = 'U' or 'u', the leading *m*-by-*m* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *m*-by-*m* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of *a* is not referenced.

Before entry with *side* = 'R' or 'r', the *n*-by-*n* part of the array *a* must contain the Hermitian matrix, such that when *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of *a* is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub) program. When *side* = 'L' or 'l' then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1,*n*).

b

COMPLEX for chemm

DOUBLE COMPLEX for zhemm

Array, size *ldb* by *n*.

The leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.*ldb* must be at least max(1, *m*)

beta

COMPLEX for chemm

DOUBLE COMPLEX for zhemm

Specifies the scalar *beta*.

When *beta* is supplied as zero, then *c* need not be set on input.

c

COMPLEX for chemm

DOUBLE COMPLEX for zhemm

Array, size (*c*, *n*). Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is zero, in which case *c* need not be set on entry.

ldc

INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program. *ldc* must be at least max(1, *m*)

Output Parameters

c

Overwritten by the *m*-by-*n* updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `hemm` interface are the following:

a	Holds the matrix *A* of size (*k*,*k*) where *k = m* if *side = 'L'*, *k = n* otherwise.
b	Holds the matrix *B* of size (*m*,*n*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be `'L'` or `'R'`. The default value is `'L'`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
alpha	The default value is 1.
beta	The default value is 0.

?herk

Performs a Hermitian rank-k update.

Syntax

```
call cherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call zherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call herk(a, c [,uplo] [, trans] [,alpha][,beta])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?herk` routines perform a rank-k matrix-matrix operation using a general matrix *A* and a Hermitian matrix *C*. The operation is defined as:

$$C := alpha*A*A^H + beta*C,$$

or

$$C := alpha*A^H*A + beta*C,$$

where:

alpha and *beta* are real scalars,

C is an *n*-by-*n* Hermitian matrix,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is used.

If *uplo* = 'U' or 'u' , then the upper triangular part of the array *c* is used.

If *uplo* = 'L' or 'l' , then the low triangular part of the array *c* is used.

trans

CHARACTER*1. Specifies the operation:

if *trans*= 'N' or 'n', then $C := alpha*A*A^H + beta*C$;

if *trans*= 'C' or 'c', then $C := alpha*A^H*A + beta*C$.

n

INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

k

INTEGER. With *trans*= 'N' or 'n', *k* specifies the number of columns of the matrix *A*, and with *trans*= 'C' or 'c', *k* specifies the number of rows of the matrix *A*.

The value of *k* must be at least zero.

alpha

REAL for cherk

DOUBLE PRECISION for zherk

Specifies the scalar *alpha*.

a

COMPLEX for cherk

DOUBLE COMPLEX for zherk

Array, size (*lda*, *ka*), where *ka* is *k* when *trans*= 'N' or 'n', and is *n* otherwise. Before entry with *trans*= 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *a*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When *trans*= 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).

beta

REAL for cherk

DOUBLE PRECISION for zherk

Specifies the scalar *beta*.

c

COMPLEX for cherk

DOUBLE COMPLEX for zherk

Array, size *ldc* by *n*.

Before entry with *uplo* = 'U' or 'u' , the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *c* is not referenced.

Before entry with *uplo* = 'L' or 'l' , the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *c* is not referenced.

The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

ldc — INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *n*).

Output Parameters

c — With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine herk interface are the following:

a — Holds the matrix *A* of size (*ma*,*ka*) where

ka = *k* if *trans*='N',

ka = *n* otherwise,

ma = *n* if *trans*='N',

ma = *k* otherwise.

c — Holds the matrix *C* of size (*n*,*n*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

trans — Must be 'N' or 'C'. The default value is 'N'.

alpha — The default value is 1.

beta — The default value is 0.

?her2k

Performs a Hermitian rank-2k update.

Syntax

```
call cher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call her2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?her2k routines perform a rank-2k matrix-matrix operation using general matrices *A* and *B* and a Hermitian matrix *C*. The operation is defined as

```
C := alpha*A*B^H + conjg(alpha)B*A^H + beta*C
```

or

```
C := alpha*A^H*B + conjg(alpha)*B^H*A + beta*C
```

where:

alpha is a scalar and *beta* is a real scalar.

C is an *n*-by-*n* Hermitian matrix.

A and *B* are *n*-by-*k* matrices in the first case and *k*-by-*n* matrices in the second case.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is used.

If *uplo* = 'U' or 'u' , then the upper triangular of the array *c* is used.

If *uplo* = 'L' or 'l' , then the low triangular of the array *c* is used.

trans

CHARACTER*1. Specifies the operation:

if*trans*= 'N' or 'n', then *C*:=*alpha***A***B*^{H} + *alpha***B***A*^{H} + *beta***C*;

if *trans*= 'C' or 'c', then *C*:=*alpha***A*^{H}**B* + *alpha***B*^{H}**A* + *beta***C*.

n

INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

k

INTEGER. With *trans*= 'N' or 'n' specifies the number of columns of the matrix *A*, and with *trans*= 'C' or 'c', *k* specifies the number of rows of the matrix *A*.

The value of *k* must be at least equal to zero.

alpha

COMPLEX for cher2k

DOUBLE COMPLEX for zher2k

Specifies the scalar *alpha*.

a

COMPLEX for cher2k

DOUBLE COMPLEX for zher2k

Array, size (*lda*, *ka*), where *ka* is *k* when *trans*= 'N' or 'n', and is *n* otherwise. Before entry with *trans*= 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

When *trans*= 'N' or 'n'*trans*= 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).

beta

REAL for cher2k

DOUBLE PRECISION for zher2k

Specifies the scalar *beta*.

b

`COMPLEX` for `cher2k`

`DOUBLE COMPLEX` for `zher2k`

Array, size `(ldb, kb)`, where *kb* is *k* when `trans= 'N'` or `'n'`, and is *n* otherwise. Before entry with `trans= 'N'` or `'n'`, the leading *n*-by-*k* part of the array *b* must contain the matrix *B*, otherwise the leading *k*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

`INTEGER`. Specifies the leading dimension of *a* as declared in the calling (sub)program.

When `trans= 'N'` or `'n'`, then *ldb* must be at least `max(1, n)`, otherwise *ldb* must be at least `max(1, k)`.

c

`COMPLEX` for `cher2k`

`DOUBLE COMPLEX` for `zher2k`

Array, size *ldc* by *n*.

Before entry with`uplo = 'U'` or `'u'`, the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *c* is not referenced.

Before entry with `uplo = 'L'` or `'l'`, the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *c* is not referenced.

The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

ldc

`INTEGER`. Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least `max(1, n)`.

Output Parameters

c

With `uplo = 'U'` or `'u'`, the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L'` or `'l'`, the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `her2k` interface are the following:

a

Holds the matrix *A* of size (*ma*,*ka*) where

ka = *k* if *trans* = `'N'`,

ka = *n* otherwise,

ma = *n* if *trans* = `'N'`,

	ma = *k* otherwise.
b	Holds the matrix *B* of size (*mb*,*kb*) where *kb* = *k* if *trans* = 'N', *kb* = *n* otherwise, *mb* = *n* if *trans* = 'N', *mb* = *k* otherwise.
c	Holds the matrix *C* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N' or 'C'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 0.

?symm

Computes a matrix-matrix product where one input matrix is symmetric.

Syntax

```
call ssymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call dsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call csymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call zsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call symm(a, b, c [,side][,uplo] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?symm routines compute a scalar-matrix-matrix product with one symmetric matrix and add the result to a scalar-matrix product . The operation is defined as

```
C := alpha*A*B + beta*C,
```

or

```
C := alpha*B*A + beta*C,
```

where:

alpha and *beta* are scalars,

A is a symmetric matrix,

B and *C* are *m*-by-*n* matrices.

Input Parameters

side

CHARACTER*1. Specifies whether the symmetric matrix *A* appears on the left or right in the operation:

if *side* = 'L' or 'l' , then *C* := *alpha***A***B* + *beta***C*;

if *side* = 'R' or 'r' , then *C* := *alpha***B***A* + *beta***C*.

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix *A* is used:

if *uplo* = 'U' or 'u', then the upper triangular part is used;

if *uplo* = 'L' or 'l', then the lower triangular part is used.

m

INTEGER. Specifies the number of rows of the matrix *C*.

The value of *m* must be at least zero.

n

INTEGER. Specifies the number of columns of the matrix *C*.

The value of *n* must be at least zero.

alpha

REAL for ssymm

DOUBLE PRECISION for dsymm

COMPLEX for csymm

DOUBLE COMPLEX for zsymm

Specifies the scalar *alpha*.

a

REAL for ssymm

DOUBLE PRECISION for dsymm

COMPLEX for csymm

DOUBLE COMPLEX for zsymm

Array, size (*lda*, *ka*) , where *ka* is *m* when *side* = 'L' or 'l' and is *n* otherwise.

Before entry with *side* = 'L' or 'l', the *m*-by-*m* part of the array *a* must contain the symmetric matrix, such that when *uplo* = 'U' or 'u', the leading *m*-by-*m* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced, and when *side* = 'L' or 'l', the leading *m*-by-*m* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

Before entry with *side* = 'R' or 'r', the *n*-by-*n* part of the array *a* must contain the symmetric matrix, such that when *uplo* = 'U' or 'u'e array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced, and when *side* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l' then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1, *n*).

b — REAL for ssymm

DOUBLE PRECISION for dsymm

COMPLEX for csymm

DOUBLE COMPLEX for zsymm

Array, size *ldb* by *n*.

The leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb — INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program. *ldb* must be at least max(1, *m*)

beta — REAL for ssymm

DOUBLE PRECISION for dsymm

COMPLEX for csymm

DOUBLE COMPLEX for zsymm

Specifies the scalar *beta*.

When *beta* is set to zero, then *c* need not be set on input.

c — REAL for ssymm

DOUBLE PRECISION for dsymm

COMPLEX for csymm

DOUBLE COMPLEX for zsymm

Array, size (*c*, *n*). Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is zero, in which case *c* need not be set on entry.

ldc — INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program. *ldc* must be at least max(1, *m*)

Output Parameters

c — Overwritten by the *m*-by-*n* updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine symm interface are the following:

a — Holds the matrix *A* of size (*k*,*k*) where

k = *m* if *side* = 'L',

k = *n* otherwise.

b	Holds the matrix *B* of size (*m*,*n*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.
beta	The default value is 0.

?syrk

Performs a symmetric rank-k update.

Syntax

```
call ssyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call dsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call csyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call zsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call syrk(a, c [,uplo] [, trans] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?syrk routines perform a rank-k matrix-matrix operation for a symmetric matrix *C* using a general matrix *A* . The operation is defined as:

```
C := alpha*A*A' + beta*C,
```

or

```
C := alpha*A'*A + beta*C,
```

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is used. If *uplo* = 'U' or 'u' , then the upper triangular part of the array *c* is used. If *uplo* = 'L' or 'l' , then the low triangular part of the array *c* is used.
trans	CHARACTER*1. Specifies the operation: if *trans*= 'N' or 'n', then *C* := *alpha*A*A'* + *beta*C*;

if *trans*= 'T' or 't', then *C* := *alpha***A*'**A* + *beta***C*;

if *trans*= 'C' or 'c', then *C* := *alpha***A*'**A* + *beta***C*.

n INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

k INTEGER. On entry with *trans*= 'N' or 'n', *k* specifies the number of columns of the matrix *a*, and on entry with *trans*= 'T', 't', 'C', or 'c' , *k* specifies the number of rows of the matrix *a*.

The value of *k* must be at least zero.

alpha REAL for ssyrk

DOUBLE PRECISION for dsyrk

COMPLEX for csyrk

DOUBLE COMPLEX for zsyrk

Specifies the scalar *alpha*.

a REAL for ssyrk

DOUBLE PRECISION for dsyrk

COMPLEX for csyrk

DOUBLE COMPLEX for zsyrk

Array, size (*lda*, *ka*), where *ka* is *k* when *trans*= 'N' or 'n', and is *n* otherwise. Before entry with *trans*= 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

Array, size (*lda*, *ka*), where *ka* is *k* when *trans*= 'N' or 'n', and is *n* otherwise. Before entry with *trans*= 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *a*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When *trans*= 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).

beta REAL for ssyrk

DOUBLE PRECISION for dsyrk

COMPLEX for csyrk

DOUBLE COMPLEX for zsyrk

Specifies the scalar *beta*.

c REAL for ssyrk

DOUBLE PRECISION for dsyrk

COMPLEX for csyrk

DOUBLE COMPLEX for zsyrk

Array, size (*ldc*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *c* is not referenced.

ldc — INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *n*).

Output Parameters

c — With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine syrk interface are the following:

a — Holds the matrix *A* of size (*ma*,*ka*) where

ka = *k* if *transa*='N',

ka = *n* otherwise,

ma = *n* if *transa*='N',

ma = *k* otherwise.

c — Holds the matrix *C* of size (*n*,*n*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

trans — Must be 'N', 'C', or 'T'.

The default value is 'N'.

alpha — The default value is 1.

beta — The default value is 0.

?syr2k

Performs a symmetric rank-2k update.

Syntax

```
call ssyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call csyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

```
call syr2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?syr2k routines perform a rank-2k matrix-matrix operation for a symmetric matrix *C* using general matrices *A* and *B*The operation is defined as:

```
C := alpha*A*B' + alpha*B*A' + beta*C,
```

or

```
C := alpha*A'*B + alpha*B'*A + beta*C,
```

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix,

A and *B* are *n*-by-*k* matrices in the first case, and *k*-by-*n* matrices in the second case.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is used. If *uplo* = 'U' or 'u' , then the upper triangular part of the array *c* is used. If *uplo* = 'L' or 'l' , then the low triangular part of the array *c* is used.
trans	CHARACTER*1. Specifies the operation: if *trans*= 'N' or 'n', then *C* := *alpha*A*B'+alpha*B*A'+beta*C*; if *trans*= 'T' or 't', then *C* := *alpha*A'*B +alpha*B'*A +beta*C*; if *trans*= 'C' or 'c', then *C* := *alpha*A'*B +alpha*B'*A +beta*C*.
n	INTEGER. Specifies the order of the matrix *C*.The value of *n* must be at least zero.
k	INTEGER. On entry with *trans*= 'N' or 'n', *k* specifies the number of columns of the matrices *A* and *B*, and on entry with *trans*= 'T' or 't' or 'C' or 'c', *k* specifies the number of rows of the matrices *A* and *B*. The value of *k* must be at least zero.
alpha	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Specifies the scalar *alpha*.
a	REAL for ssyr2k DOUBLE PRECISION for dsyr2k

COMPLEX for csyr2k

DOUBLE COMPLEX for zsyr2k

Array, size (*lda*, *ka*), where *ka* is *k* when *trans*= 'N' or 'n', and is *n* otherwise. Before entry with *trans*= 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

When *trans*= 'N' or 'n'*trans*= 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).

b REAL for ssyr2k

DOUBLE PRECISION for dsyr2k

COMPLEX for csyr2k

DOUBLE COMPLEX for zsyr2k

Array, size (*ldb*, *kb*), where *kb* is *k* when *trans*= 'N' or 'n', and is *n* otherwise. Before entry with *trans*= 'N' or 'n', the leading *n*-by-*k* part of the array *b* must contain the matrix *B*, otherwise the leading *k*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

When *trans*= 'N' or 'n', then *ldb* must be at least max(1, *n*), otherwise *ldb* must be at least max(1, *k*).

beta REAL for ssyr2k

DOUBLE PRECISION for dsyr2k

COMPLEX for csyr2k

DOUBLE COMPLEX for zsyr2k

Specifies the scalar *beta*.

c REAL for ssyr2k

DOUBLE PRECISION for dsyr2k

COMPLEX for csyr2k

DOUBLE COMPLEX for zsyr2k

Array, size (*ldc*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *c* is not referenced.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *n*).

Output Parameters

c	With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine syr2k interface are the following:

a	Holds the matrix *A* of size (*ma*,*ka*) where *ka* = *k* if *trans* = 'N', *ka* = *n* otherwise, *ma* = *n* if *trans* = 'N', *ma* = *k* otherwise.
b	Holds the matrix *B* of size (*mb*,*kb*) where *kb* = *k* if *trans* = 'N', *kb* = *n* otherwise, *mb* = *n* if *trans* = 'N', *mb* = *k* otherwise.
c	Holds the matrix *C* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 0.

?trmm

Computes a matrix-matrix product where one input matrix is triangular.

Syntax

```
call strmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call dtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ctrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ztrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call trmm(a, b [,side] [, uplo] [,transa][,diag] [,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?trmm routines compute a scalar-matrix-matrix product with one triangular matrix . The operation is defined as

```
B := alpha*op(A)*B
```

or

```
B := alpha*B*op(A)
```

where:

alpha is a scalar,

B is an *m*-by-*n* matrix,

A is a unit, or non-unit, upper or lower triangular matrix

op(*A*) is one of op(*A*) = *A*, or op(A) = *A*', or op(*A*) = conjg(*A*').

Input Parameters

side	CHARACTER*1. Specifies whether op(*A*) appears on the left or right of *B* in the operation: if *side* = 'L' or 'l' , then *B* := *alpha**op(*A*)**B*; if *side* = 'R' or 'r' , then *B* := *alpha***B**op(*A*).
uplo	CHARACTER*1. Specifies whether the matrix *A* is upper or lower triangular. *uplo* = 'U' or 'u' if *uplo* = 'L' or 'l', then the matrix is low triangular.
transa	CHARACTER*1. Specifies the form of op(*A*) used in the matrix multiplication: if *transa*= 'N' or 'n' , then op(*A*) = *A*; if *transa*= 'T' or 't' , then op(*A*) = *A*'; if *transa*= 'C' or 'c' , then op(*A*) = conjg(*A*').
diag	CHARACTER*1. Specifies whether the matrix *A* is unit triangular: if *diag* = 'U' or 'u' then the matrix is unit triangular; if *diag* = 'N' or 'n' , then the matrix is not unit triangular.
m	INTEGER. Specifies the number of rows of *B*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of *B*. The value of *n* must be at least zero.
alpha	REAL for strmm DOUBLE PRECISION for dtrmm

COMPLEX for ctrmm

DOUBLE COMPLEX for ztrmm

Specifies the scalar *alpha*.

When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.

a

REAL for strmm

DOUBLE PRECISION for dtrmm

COMPLEX for ctrmm

DOUBLE COMPLEX for ztrmm

Array, size *lda* by *k*, where *k* is *m* when *side* = 'L'side or 'l' and is *n* when *side* = 'R' or 'r'. Before entry with *uplo* = 'U' or 'u', the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *k* by *k* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced.

When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When*side* = 'L' or 'l', then *lda* must be at least max(1, *m*), when *side* = 'R' or 'r', then *lda* must be at least max(1, *n*).

b

REAL for strmm

DOUBLE PRECISION for dtrmm

COMPLEX for ctrmm

DOUBLE COMPLEX for ztrmm

Array, size *ldb* by *n*. Before entry, the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program. *ldb* must be at least max(1, *m*)

Output Parameters

b

Overwritten by the transformed matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine trmm interface are the following:

a

Holds the matrix *A* of size (*k*,*k*) where

k = *m* if *side* = 'L',

	k = *n* otherwise.
b	Holds the matrix *B* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
transa	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.
alpha	The default value is 1.

?trsm

Solves a triangular matrix equation.

Syntax

```
call strsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call dtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ctrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ztrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call trsm(a, b [,side] [, uplo] [,transa][,diag] [,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?trsm routines solve one of the following matrix equations:

```
op(A)*X = alpha*B,
```

or

```
X*op(A) = alpha*B,
```

where:

alpha is a scalar,

X and *B* are *m*-by-*n* matrices,

A is a unit, or non-unit, upper or lower triangular matrix, and

op(*A*) is one of op(*A*) = *A*, or op(*A*) = *A*', or op(*A*) = conjg(*A*').

The matrix *B* is overwritten by the solution matrix *X*.

Input Parameters

side	CHARACTER*1. Specifies whether op(*A*) appears on the left or right of *X* in the equation: if *side* = 'L' or 'l' , then op(*A*)**X* = *alpha***B*;

if *side* = 'R' or 'r' , then *X*op(*A*) = *alpha***B*.

uplo — CHARACTER*1. Specifies whether the matrix *A* is upper or lower triangular.

uplo = 'U' or 'u'
if *uplo* = 'L' or 'l', then the matrix is low triangular.

transa — CHARACTER*1. Specifies the form of op(*A*) used in the matrix multiplication:

if *transa*= 'N' or 'n' , then op(*A*) = *A*;

if *transa*= 'T' or 't' ;

if *transa*= 'C' or 'c' , then op(*A*) = conjg(*A*').

diag — CHARACTER*1. Specifies whether the matrix *A* is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n' , then the matrix is not unit triangular.

m — INTEGER. Specifies the number of rows of *B*. The value of *m* must be at least zero.

n — INTEGER. Specifies the number of columns of *B*. The value of *n* must be at least zero.

alpha — REAL for strsm

DOUBLE PRECISION for dtrsm

COMPLEX for ctrsm

DOUBLE COMPLEX for ztrsm

Specifies the scalar *alpha*.

When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.

a — REAL for strsm

DOUBLE PRECISION for dtrsm

COMPLEX for ctrsm

DOUBLE COMPLEX for ztrsm

Array, size (*lda*, *k*) , where *k* is *m* when *side* = 'L' or 'l' and is *n* when *side* = 'R' or 'r' . Before entry with *uplo* = 'U' or 'u' , the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l' lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced.

When *diag* = 'U' or 'u' , the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda	INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l' , then *lda* must be at least max(1, *m*), when *side* = 'R' or 'r' , then *lda* must be at least max(1, *n*).
b	REAL for strsm DOUBLE PRECISION for dtrsm COMPLEX for ctrsm DOUBLE COMPLEX for ztrsm Array, size *ldb* by *n*. Before entry, the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program. *ldb* must be at least max(1, *m*) .

Output Parameters

b	Overwritten by the solution matrix *X*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine trsm interface are the following:

a	Holds the matrix *A* of size (*k*,*k*) where *k* = *m* if *side* = 'L', *k* = *n* otherwise.
b	Holds the matrix *B* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
transa	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.
alpha	The default value is 1.

Sparse BLAS Level 1 Routines

This section describes Sparse BLAS Level 1, an extension of BLAS Level 1 included in the Intel® oneAPI Math Kernel Library beginning with the Intel® oneAPI Math Kernel Library (oneMKL) release 2.1. Sparse BLAS Level 1 is a group of routines and functions that perform a number of common vector operations on sparse vectors stored in compressed form.

Sparse vectors are those in which the majority of elements are zeros. Sparse BLAS routines and functions are specially implemented to take advantage of vector sparsity. This allows you to achieve large savings in computer time and memory. If *nz* is the number of non-zero vector elements, the computer time taken by Sparse BLAS operations will be *O*(*nz*).

Vector Arguments

Compressed sparse vectors. Let *a* be a vector stored in an array, and assume that the only non-zero elements of *a* are the following:

$a(k_1), a(k_2), a(k_3) \ldots a(k_{nz}),$

where *nz* is the total number of non-zero elements in *a*.

In Sparse BLAS, this vector can be represented in compressed form by two arrays, *x* (values) and *indx* (indices). Each array has *nz* elements:

$x(1)=a(k_1), x(2)=a(k_2), \ldots x(nz)= a(k_{nz}),$

$indx(1)=k_1, indx(2)=k_2, \ldots indx(nz)= k_{nz}.$

Thus, a sparse vector is fully determined by the triple (*nz*, *x*, *indx*). If you pass a negative or zero value of *nz* to Sparse BLAS, the subroutines do not modify any arrays or variables.

Full-storage vectors. Sparse BLAS routines can also use a vector argument fully stored in a single array (a full-storage vector). If *y* is a full-storage vector, its elements must be stored contiguously: the first element in `y(1)`, the second in `y(2)`, and so on. This corresponds to an increment `incy = 1` in BLAS Level 1. No increment value for full-storage vectors is passed as an argument to Sparse BLAS routines or functions.

Naming Conventions for Sparse BLAS Routines

Similar to BLAS, the names of Sparse BLAS subprograms have prefixes that determine the data type involved: *s* and *d* for single- and double-precision real; *c* and *z* for single- and double-precision complex respectively.

If a Sparse BLAS routine is an extension of a "dense" one, the subprogram name is formed by appending the suffix *i* (standing for *indexed*) to the name of the corresponding "dense" subprogram. For example, the Sparse BLAS routine `saxpyi` corresponds to the BLAS routine `saxpy`, and the Sparse BLAS function `cdotci` corresponds to the BLAS function `cdotc`.

Routines and Data Types

Routines and data types supported in the Intel® oneAPI Math Kernel Library (oneMKL) implementation of Sparse BLAS are listed inTable “Sparse BLAS Routines and Their Data Types”.

Sparse BLAS Routines and Their Data Types

Routine/ Function	Data Types	Description
`?axpyi`	s, d, c, z	Scalar-vector product plus vector (routines)
`?doti`	s, d	Dot product (functions)
`?dotci`	c, z	Complex dot product conjugated (functions)
`?dotui`	c, z	Complex dot product unconjugated (functions)
`?gthr`	s, d, c, z	Gathering a full-storage sparse vector into compressed form *nz*, *x*, *indx* (routines)
`?gthrz`	s, d, c, z	Gathering a full-storage sparse vector into compressed form and assigning zeros to gathered elements in the full-storage vector (routines)
`?roti`	s, d	Givens rotation (routines)
`?sctr`	s, d, c, z	Scattering a vector from compressed form to full-storage form (routines)

BLAS Level 1 Routines That Can Work With Sparse Vectors

The following BLAS Level 1 routines will give correct results when you pass to them a compressed-form array *x*(with the increment *incx*=1):

?asum	sum of absolute values of vector elements
?copy	copying a vector
?nrm2	Euclidean norm of a vector
?scal	scaling a vector
i?amax	index of the element with the largest absolute value for real flavors, or the largest sum |Re(*x*(*i*))|+|Im(*x*(*i*))| for complex flavors.
i?amin	index of the element with the smallest absolute value for real flavors, or the smallest sum |Re(*x*(*i*))|+|Im(*x*(*i*))| for complex flavors.

The result *i* returned by i?amax and i?amin should be interpreted as index in the compressed-form array, so that the largest (smallest) value is *x*(*i*); the corresponding index in full-storage array is *indx*(*i*).

You can also call ?rotg to compute the parameters of Givens rotation and then pass these parameters to the Sparse BLAS routines ?roti.

?axpyi

Adds a scalar multiple of compressed sparse vector to a full-storage vector.

Syntax

```
call saxpyi(nz, a, x, indx, y)
call daxpyi(nz, a, x, indx, y)
call caxpyi(nz, a, x, indx, y)
call zaxpyi(nz, a, x, indx, y)
call axpyi(x, indx, y [, a])
```

Include Files

- mkl.fi, blas.f90

Description

The ?axpyi routines perform a vector-vector operation defined as

```
y := a*x + y
```

where:

a is a scalar,

x is a sparse vector stored in compressed form,

y is a vector in full storage form.

The ?axpyi routines reference or modify only the elements of *y* whose indices are listed in the array *indx*.

The values in *indx* must be distinct.

Input Parameters

nz	INTEGER. The number of elements in *x* and *indx*.
a	REAL for saxpyi DOUBLE PRECISION for daxpyi COMPLEX for caxpyi DOUBLE COMPLEX for zaxpyi Specifies the scalar *a*.
x	REAL for saxpyi DOUBLE PRECISION for daxpyi COMPLEX for caxpyi DOUBLE COMPLEX for zaxpyi Array, size at least *nz*.
indx	INTEGER. Specifies the indices for the elements of *x*. Array, size at least *nz*.
y	REAL for saxpyi DOUBLE PRECISION for daxpyi COMPLEX for caxpyi DOUBLE COMPLEX for zaxpyi Array, size at least max(*indx*(i)).

Output Parameters

y	Contains the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine axpyi interface are the following:

x	Holds the vector with the number of elements *nz*.
indx	Holds the vector with the number of elements *nz*.
y	Holds the vector with the number of elements *nz*.
a	The default value is 1.

?doti

Computes the dot product of a compressed sparse real vector by a full-storage real vector.

Syntax

res = sdoti(*nz*, *x*, *indx*, *y*)

res = ddoti(*nz*, *x*, *indx*, *y*)

res = doti(*x*, *indx*, *y*)

Include Files

- mkl.fi, blas.f90

Description

The ?doti routines return the dot product of *x* and *y* defined as

```
res = x(1)*y(indx(1)) + x(2)*y(indx(2)) +...+ x(nz)*y(indx(nz))
```

where the triple (*nz*, *x*, *indx*) defines a sparse real vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	INTEGER. The number of elements in *x* and *indx* .
x	REAL for sdoti DOUBLE PRECISION for ddoti Array, size at least *nz*.
indx	INTEGER. Specifies the indices for the elements of *x*. Array, size at least *nz*.
y	REAL for sdoti DOUBLE PRECISION for ddoti Array, size at least max(*indx(i)*).

Output Parameters

res	REAL for sdoti DOUBLE PRECISION for ddoti Contains the dot product of *x* and *y*, if *nz* is positive. Otherwise, *res* contains 0.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine doti interface are the following:

x	Holds the vector with the number of elements *nz*.
indx	Holds the vector with the number of elements *nz*.

y	Holds the vector with the number of elements *nz*.

?dotci

Computes the conjugated dot product of a compressed sparse complex vector with a full-storage complex vector.

Syntax

```
res = cdotci(nz, x, indx, y )
res = zdotci(nzz, x, indx, y )
res = dotci(x, indx, y)
```

Include Files

- mkl.fi, blas.f90

Description

The ?dotci routines return the dot product of *x* and *y* defined as

```
conjg(x(1))*y(indx(1)) + ... + conjg(x(nz))*y(indx(nz))
```

where the triple (*nz*, *x*, *indx*) defines a sparse complex vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	INTEGER. The number of elements in *x* and *indx* .
x	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Array, size at least *nz*.
indx	INTEGER. Specifies the indices for the elements of *x*. Array, size at least *nz*.
y	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Array, size at least max(*indx(i)*).

Output Parameters

res	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Contains the conjugated dot product of *x* and *y*, if *nz* is positive. Otherwise, it contains 0.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `dotci` interface are the following:

x	Holds the vector with the number of elements (*nz*).
indx	Holds the vector with the number of elements (*nz*).
y	Holds the vector with the number of elements (*nz*).

?dotui

Computes the dot product of a compressed sparse complex vector by a full-storage complex vector.

Syntax

```
res = cdotui(nz, x, indx, y )
res = zdotui(nzz, x, indx, y )
res = dotui(x, indx, y)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?dotui` routines return the dot product of *x* and *y* defined as

```
res = x(1)*y(indx(1)) + x(2)*y(indx(2)) +...+ x(nz)*y(indx(nz))
```

where the triple (*nz*, *x*, *indx*) defines a sparse complex vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	`INTEGER`. The number of elements in *x* and *indx*.

x	`COMPLEX` for `cdotui`
	`DOUBLE COMPLEX` for `zdotui`
	Array, size at least *nz*.

indx	`INTEGER`. Specifies the indices for the elements of *x*. Array, size at least *nz*.

y	`COMPLEX` for `cdotui`
	`DOUBLE COMPLEX` for `zdotui`
	Array, size at least max(*indx(i)*).

Output Parameters

res	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Contains the dot product of *x* and *y*, if *nz* is positive. Otherwise, *res* contains 0.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine dotui interface are the following:

x	Holds the vector with the number of elements *nz*.
indx	Holds the vector with the number of elements *nz*.
y	Holds the vector with the number of elements *nz*.

?gthr

Gathers a full-storage sparse vector's elements into compressed form.

Syntax

```
call sgthr(nz, y, x, indx )
call dgthr(nz, y, x, indx )
call cgthr(nz, y, x, indx )
call zgthr(nz, y, x, indx )
res = gthr(x, indx, y)
```

Include Files

- mkl.fi, blas.f90

Description

The ?gthr routines gather the specified elements of a full-storage sparse vector *y* into compressed form(*nz*, *x*, *indx*). The routines reference only the elements of *y* whose indices are listed in the array *indx*:

x(*i*) = *y*(*indx*(*i*)), for *i*=1,2,... ,*nz*.

Input Parameters

nz	INTEGER. The number of elements of *y* to be gathered.
indx	INTEGER. Specifies indices of elements to be gathered. Array, size at least *nz*.
y	REAL for sgthr DOUBLE PRECISION for dgthr

	`COMPLEX` for `cgthr`
	`DOUBLE COMPLEX` for `zgthr`
	Array, size at least max(*indx(i)*).

Output Parameters

x	`REAL` for `sgthr`
	`DOUBLE PRECISION` for `dgthr`
	`COMPLEX` for `cgthr`
	`DOUBLE COMPLEX` for `zgthr`
	Array, size at least *nz*.
	Contains the vector converted to the compressed form.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `gthr` interface are the following:

x	Holds the vector with the number of elements *nz*.
indx	Holds the vector with the number of elements *nz*.
y	Holds the vector with the number of elements *nz*.

?gthrz

Gathers a sparse vector's elements into compressed form, replacing them by zeros.

Syntax

```
call sgthrz(nz, y, x, indx )
call dgthrz(nz, y, x, indx )
call cgthrz(nz, y, x, indx )
call zgthrz(nz, y, x, indx )
res = gthrz(x, indx, y)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?gthrz` routines gather the elements with indices specified by the array *indx* from a full-storage vector *y* into compressed form (*nz*, *x*, *indx*) and overwrite the gathered elements of *y* by zeros. Other elements of *y* are not referenced or modified (see also `?gthr`).

Input Parameters

nz	INTEGER. The number of elements of *y* to be gathered.
indx	INTEGER. Specifies indices of elements to be gathered. Array, size at least *nz*.
y	REAL for sgthrz DOUBLE PRECISION for dgthrz COMPLEX for cgthrz DOUBLE COMPLEX for zgthrz Array, size at least max(*indx(i)*).

Output Parameters

x	REAL for sgthrz DOUBLE PRECISION for dgthrz COMPLEX for cgthrz DOUBLE COMPLEX for zgthrz Array, size at least *nz*. Contains the vector converted to the compressed form.
y	The updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine gthrz interface are the following:

x	Holds the vector with the number of elements *nz*.
indx	Holds the vector with the number of elements *nz*.
y	Holds the vector with the number of elements *nz*.

?roti

Applies Givens rotation to sparse vectors one of which is in compressed form.

Syntax

```
call sroti(nz, x, indx, y, c, s)
call droti(nz, x, indx, y, c, s)
call roti(x, indx, y, c, s)
```

Include Files

- mkl.fi, blas.f90

Description

The ?roti routines apply the Givens rotation to elements of two real vectors, *x* (in compressed form *nz*, *x*, *indx*) and *y* (in full storage form):

```
x(i) = c*x(i) + s*y(indx(i))
```

```
y(indx(i)) = c*y(indx(i))- s*x(i)
```

The routines reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	INTEGER. The number of elements in *x* and *indx*.
x	REAL for sroti DOUBLE PRECISION for droti Array, size at least *nz*.
indx	INTEGER. Specifies the indices for the elements of *x*. Array, size at least *nz*.
y	REAL for sroti DOUBLE PRECISION for droti Array, size at least max(*indx(i)*).
c	REAL for sroti DOUBLE PRECISION for droti. A scalar.
s	REAL for sroti DOUBLE PRECISION for droti. A scalar.

Output Parameters

x and *y*	The updated arrays.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine roti interface are the following:

x	Holds the vector with the number of elements *nz*.
indx	Holds the vector with the number of elements *nz*.
y	Holds the vector with the number of elements *nz*.

?sctr

Converts compressed sparse vectors into full storage form.

Syntax

```
call ssctr(nz, x, indx, y )
call dsctr(nz, x, indx, y )
call csctr(nz, x, indx, y )
call zsctr(nz, x, indx, y )
call sctr(x, indx, y)
```

Include Files

- mkl.fi, blas.f90

Description

The ?sctr routines scatter the elements of the compressed sparse vector (*nz*, *x*, *indx*) to a full-storage vector *y*. The routines modify only the elements of *y* whose indices are listed in the array *indx*:

y(*indx*(*i*)) = *x*(*i*), for *i*=1,2,... ,*nz*.

Input Parameters

nz	INTEGER. The number of elements of *x* to be scattered.
indx	INTEGER. Specifies indices of elements to be scattered. Array, size at least *nz*.
x	REAL for ssctr DOUBLE PRECISION for dsctr COMPLEX for csctr DOUBLE COMPLEX for zsctr Array, size at least *nz*. Contains the vector to be converted to full-storage form.

Output Parameters

y	REAL for ssctr DOUBLE PRECISION for dsctr COMPLEX for csctr DOUBLE COMPLEX for zsctr Array, size at least max(*indx(i)*). Contains the vector *y* with updated elements.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `sctr` interface are the following:

x	Holds the vector with the number of elements *nz*.
indx	Holds the vector with the number of elements *nz*.
y	Holds the vector with the number of elements *nz*.

Sparse BLAS Level 2 and Level 3 Routines

> **NOTE** The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines are deprecated. Use the corresponding routine from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface as indicated in the description for each routine.

This section describes Sparse BLAS Level 2 and Level 3 routines included in the Intel® oneAPI Math Kernel Library (oneMKL) . Sparse BLAS Level 2 is a group of routines and functions that perform operations between a sparse matrix and dense vectors. Sparse BLAS Level 3 is a group of routines and functions that perform operations between a sparse matrix and dense matrices.

The terms and concepts required to understand the use of the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines are discussed in theLinear Solvers Basics appendix.

The Sparse BLAS routines can be useful to implement iterative methods for solving large sparse systems of equations or eigenvalue problems. For example, these routines can be considered as building blocks for Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS).

Intel® oneAPI Math Kernel Library (oneMKL) provides Sparse BLAS Level 2 and Level 3 routines with typical (or conventional) interface similar to the interface used in the NIST* Sparse BLAS library [Rem05].

Some software packages and libraries (the PARDISO* Solverused in Intel® oneAPI Math Kernel Library (oneMKL),*Sparskit 2* [Saad94], the Compaq* Extended Math Library (CXML)[CXML01]) use different (early) variation of the compressed sparse row (CSR) format and support only Level 2 operations with simplified interfaces. Intel® oneAPI Math Kernel Library (oneMKL) provides an additional set of Sparse BLAS Level 2 routines with similar simplified interfaces. Each of these routines operates only on a matrix of the fixed type.

The routines described in this section support both one-based indexing and zero-based indexing of the input data (see details in the section One-based and Zero-based Indexing).

Naming Conventions in Sparse BLAS Level 2 and Level 3

Each Sparse BLAS Level 2 and Level 3 routine has a six- or eight-character base name preceded by the prefix `mkl_` or `mkl_cspblas_` .

The routines with typical (conventional) interface have six-character base names in accordance with the template:

```
mkl_<character > <data> <operation>( )
```

The routines with simplified interfaces have eight-character base names in accordance with the templates:

```
mkl_<character > <data> <mtype> <operation>( )
```

for routines with one-based indexing; and

```
mkl_cspblas_<character> <data><mtype><operation>( )
```

for routines with zero-based indexing.

The <character> field indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

The <data> field indicates the sparse matrix storage format (see section Sparse Matrix Storage Formats):

coo	coordinate format
csr	compressed sparse row format and its variations
csc	compressed sparse column format and its variations
dia	diagonal format
sky	skyline storage format
bsr	block sparse row format and its variations

The <operation> field indicates the type of operation:

mv	matrix-vector product (Level 2)
mm	matrix-matrix product (Level 3)
sv	solving a single triangular system (Level 2)
sm	solving triangular systems with multiple right-hand sides (Level 3)

The field <mtype> indicates the matrix type:

ge	sparse representation of a general matrix
sy	sparse representation of the upper or lower triangle of a symmetric matrix
tr	sparse representation of a triangular matrix

Sparse Matrix Storage Formats for Sparse BLAS Routines

The current version of Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines support the following point entry [Duff86] storage formats for sparse matrices:

- *compressed sparse row* format (CSR) and its variations;
- *compressed sparse column* format (CSC);
- *coordinate* format;
- *diagonal* format;
- *skyline* storage format;

and one block entry storage format:

- *block sparse row* format (BSR) and its variations.

For more information see "Sparse Matrix Storage Formats" in the Appendix"Linear Solvers Basics".

Intel® oneAPI Math Kernel Library (oneMKL) provides auxiliary routines -matrix converters - that convert sparse matrix from one storage format to another.

Routines and Supported Operations

This section describes operations supported by the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines. The following notations are used here:

A is a sparse matrix;
B and *C* are dense matrices;
D is a diagonal scaling matrix;
x and *y* are dense vectors;
alpha and *beta* are scalars;

`op(A)` is one of the possible operations:

`op(A)` = *A*;
`op(A)` = A^T - transpose of *A*;
`op(A)` = A^H - conjugated transpose of *A*.

`inv(op(A))` denotes the inverse of `op(A)`.

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines support the following operations:

- computing the vector product between a sparse matrix and a dense vector:

```
y := alpha*op(A)*x + beta*y
```

- solving a single triangular system:

```
y := alpha*inv(op(A))*x
```

- computing a product between sparse matrix and dense matrix:

```
C := alpha*op(A)*B + beta*C
```

- solving a sparse triangular system with multiple right-hand sides:

```
C := alpha*inv(op(A))*B
```

Intel® oneAPI Math Kernel Library (oneMKL) provides an additional set of the Sparse BLAS Level 2 routines with*simplified interfaces*. Each of these routines operates on a matrix of the fixed type. The following operations are supported:

- computing the vector product between a sparse matrix and a dense vector (for general and symmetric matrices):

```
y := op(A)*x
```

- solving a single triangular system (for triangular matrices):

```
y := inv(op(A))*x
```

Matrix type is indicated by the field <*mtype*> in the routine name (see section Naming Conventions in Sparse BLAS Level 2 and Level 3).

NOTE
The routines with simplified interfaces support only four sparse matrix storage formats, specifically:

CSR format in the 3-array variation accepted in the direct sparse solvers and in the CXML;
diagonal format accepted in the CXML;
coordinate format;
BSR format in the 3-array variation.

Note that routines with both typical (conventional) and simplified interfaces use the same computational kernels that work with certain internal data structures.

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines do not support in-place operations.

Complete list of all routines is given in the "Sparse BLAS Level 2 and Level 3 Routines".

Interface Consideration

One-Based and Zero-Based Indexing

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines support one-based and zero-based indexing of data arrays.

Routines with typical interfaces support zero-based indexing for the following sparse data storage formats: CSR, CSC, BSR, and COO. Routines with simplified interfaces support zero based indexing for the following sparse data storage formats: CSR, BSR, and COO. See the complete list of Sparse BLAS Level 2 and Level 3 Routines.

The one-based indexing uses the convention of starting array indices at 1. The zero-based indexing uses the convention of starting array indices at 0. For example, indices of the 5-element array *x* can be presented in case of one-based indexing as follows:

Element index: `1 2 3 4 5`

Element value: `1.0 5.0 7.0 8.0 9.0`

and in case of zero-based indexing as follows:

Element index: `0 1 2 3 4`

Element value: `1.0 5.0 7.0 8.0 9.0`

The detailed descriptions of the one-based and zero-based variants of the sparse data storage formats are given in the "Sparse Matrix Storage Formats" in the Appendix "Linear Solvers Basics".

Most parameters of the routines are identical for both one-based and zero-based indexing, but some of them have certain differences. The following table lists all these differences.

Parameter	One-based Indexing	Zero-based Indexing
val	Array containing non-zero elements of the matrix *A*, its length is *pntre*(m) – *pntrb*(1).	Array containing non-zero elements of the matrix *A*, its length is *pntre*(m–1) – *pntrb*(0).
pntrb	Array of length *m*. This array contains row indices, such that *pntrb*(i) – *pntrb*(1)+1 is the first index of row i in the arrays *val* and *indx*	Array of length *m*. This array contains row indices, such that *pntrb*(i) – *pntrb*(0) is the first index of row i in the arrays *val* and *indx*.
pntre	Array of length *m*. This array contains row indices, such that *pntre*(*I*) – *pntrb*(1) is the last index of row i in the arrays *val* and *indx*.	Array of length *m*. This array contains row indices, such that *pntre*(i) – *pntrb*(0)–1 is the last index of row i in the arrays *val* and *indx*.
ia	Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(i) is the index in the array *a* of the first non-zero element from the row i. The value of the last element *ia*(*m* + 1) is equal to the number of non-zeros plus one.	Array of length *m*+1, containing indices of elements in the array *a*, such that *ia*(i) is the index in the array *a* of the first non-zero element from the row i. The value of the last element *ia*(*m*) is equal to the number of non-zeros.
ldb	Specifies the leading dimension of *b* as declared in the calling (sub)program.	Specifies the second dimension of *b* as declared in the calling (sub)program.

Parameter	One-based Indexing	Zero-based Indexing
ldc	Specifies the leading dimension of *c* as declared in the calling (sub)program.	Specifies the second dimension of *c* as declared in the calling (sub)program.

Differences Between Intel MKL and NIST* Interfaces

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 3 routines have the following conventional interfaces:

mkl_xyyymm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, arg(*A*), *b*, *ldb*, *beta*, *c*, *ldc*), for matrix-matrix product;

mkl_xyyysm(*transa*, *m*, *n*, *alpha*, *matdescra*, arg(*A*), *b*, *ldb*, *c*, *ldc*), for triangular solvers with multiple right-hand sides.

Here x denotes data type, and *yyy* - sparse matrix data structure (storage format).

The analogous NIST* Sparse BLAS (NSB) library routines have the following interfaces:

xyyymm(*transa*, *m*, *n*, *k*, *alpha*, *descra*, arg(*A*), *b*, *ldb*, *beta*, *c*, *ldc*, *work*, *lwork*), for matrix-matrix product;

xyyysm(*transa*, *m*, *n*, *unitd*, *dv*, *alpha*, *descra*, arg(*A*), *b*, *ldb*, *beta*, *c*, *ldc*, *work*, *lwork*), for triangular solvers with multiple right-hand sides.

Some similar arguments are used in both libraries. The argument *transa* indicates what operation is performed and is slightly different in the NSB library (see Table "Parameter *transa*"). The arguments *m* and *k* are the number of rows and column in the matrix *A*, respectively, *n* is the number of columns in the matrix *C*. The arguments *alpha* and *beta* are scalar *alpha* and *beta* respectively (*beta*is not used in the Intel® oneAPI Math Kernel Library (oneMKL) triangular solvers.) The arguments*b* and *c* are rectangular arrays with the leading dimension *ldb* and *ldc*, respectively. arg(*A*) denotes the list of arguments that describe the sparse representation of *A*.

Parameter *transa*

	MKL interface	NSB interface	Operation
data type	CHARACTER*1	INTEGER	
value	N or n	0	op(*A*) = *A*
	T or t	1	op(*A*) = A^T
	C or c	2	op(*A*) = A^T or op(*A*) = A^H

Parameter matdescra

The parameter *matdescra* describes the relevant characteristic of the matrix *A*. This manual describes *matdescra*as an array of six elements in line with the NIST* implementation. However, only the first four elements of the array are used in the current versions of the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines. Elements*matdescra*(5) and *matdescra*(6) are reserved for future use. Note that whether *matdescra*is described in your application as an array of length 6 or 4 is of no importance because the array is declared as a pointer in the Intel® oneAPI Math Kernel Library (oneMKL) routines. To learn more about declaration of the*matdescra*array, see the Sparse BLAS examples located in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:examples/spblasf/ for Fortran . The table below lists elements of the parameter *matdescra*, their Fortran values, and their meanings. The parameter *matdescra* corresponds to the argument *descra* from NSB library.

Possible Values of the Parameter *matdescra (descra)*

	MKL interface		NSB interface	Matrix characteristics
	one-based indexing	zero-based indexing		
data type	CHARACTER	Char	INTEGER	
1st element	*matdescra*(1)	*matdescra*(0)	*descra*(1)	matrix structure
value	G	G	0	general
	S	S	1	symmetric ($A = A^T$)
	H	H	2	Hermitian ($A = (A^H)$)
	T	T	3	triangular
	A	A	4	skew(anti)-symmetric ($A = -A^T$)
	D	D	5	diagonal
2nd element	*matdescra*(2)	*matdescra*(1)	*descra*(2)	upper/lower triangular indicator
value	L	L	1	lower
	U	U	2	upper
3rd element	*matdescra*(3)	*matdescra*(2)	*descra*(3)	main diagonal type
value	N	N	0	non-unit
	U	U	1	unit
4th element	*matdescra*(4)	*matdescra*(3)	*descra*(4)	type of indexing
value	F		1	one-based indexing
		C	0	zero-based indexing

In some cases possible element values of the parameter matdescra depend on the values of other elements. The Table "Possible Combinations of Element Values of the Parameter *matdescra*" lists all possible combinations of element values for both multiplication routines and triangular solvers.

Possible Combinations of Element Values of the Parameter *matdescra*

Routines	matdescra(1)	matdescra(2)	matdescra(3)	matdescra(4)
Multiplication Routines	G	ignored	ignored	**F** (default) or C
	S or H	**L** (default)	**N** (default)	**F** (default) or C
	S or H	**L** (default)	U	**F** (default) or C
	S or H	U	**N** (default)	**F** (default) or C
	S or H	U	U	**F** (default) or C
	A	**L** (default)	ignored	**F** (default) or C
	A	U	ignored	**F** (default) or C

Routines	matdescra(1)	matdescra(2)	matdescra(3)	matdescra(4)
Multiplication Routines and Triangular Solvers	T	L	U	**F** (default) or C
	T	L	N	**F** (default) or C
	T	U	U	**F** (default) or C
	T	U	N	**F** (default) or C
	D	ignored	**N** (default)	**F** (default) or C
	D	ignored	U	**F** (default) or C

For a matrix in the skyline format with the main diagonal declared to be a unit, diagonal elements must be stored in the sparse representation even if they are zero. In all other formats, diagonal elements can be stored (if needed) in the sparse representation if they are not zero.

Operations with Partial Matrices

One of the distinctive feature of the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines is a possibility to perform operations only on partial matrices composed of certain parts (triangles and the main diagonal) of the input sparse matrix. It can be done by setting properly first three elements of the parameter`matdescra`.

An arbitrary sparse matrix *A* can be decomposed as

```
A = L + D + U
```

where *L* is the strict lower triangle of *A*, *U* is the strict upper triangle of *A*, *D* is the main diagonal.

Table "Output Matrices for Multiplication Routines" shows correspondence between the output matrices and values of the parameter `matdescra` for the sparse matrix *A* for multiplication routines.

Output Matrices for Multiplication Routines

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
G	ignored	ignored	`alpha*op(A)*x + beta*y` *alpha**op(*A*)**B* + *beta***C*
S or H	L	N	`alpha*op(L+D+L')*x + beta*y` *alpha**op(*L*+*D*+*L*')**B* + *beta***C*
S or H	L	U	`alpha*op(L+I+L')*x + beta*y` `alpha*op(L+I+L')*B + beta*C`
S or H	U	N	`alpha*op(U'+D+U)*x + beta*y` *alpha**op(*U*'+*D*+*U*)**B* + *beta***C*
S or H	U	U	`alpha*op(U'+I+U)*x + beta*y` *alpha**op(*U*'+*I*+*U*)**B* + *beta***C*
T	L	U	`alpha*op(L+I)*x + beta*y` *alpha**op(*L*+*I*)**B* + *beta***C*
T	L	N	`alpha*op(L+D)*x + beta*y` *alpha**op(*L*+*D*)**B* + *beta***C*
T	U	U	`alpha*op(U+I)*x + beta*y`

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
			*alpha**op(*U+I*)**B* + *beta***C*
T	U	N	`alpha*op(U+D)*x + beta*y` *alpha**op(*U+D*)**B* + *beta***C*
A	L	ignored	`alpha*op(L-L')*x + beta*y` *alpha**op(*L-L'*)**B* + *beta***C*
A	U	ignored	`alpha*op(U-U')*x + beta*y` *alpha**op(*U-U'*)**B* + *beta***C*
D	ignored	N	`alpha*D*x + beta*y` *alpha***D***B* + *beta***C*
D	ignored	U	`alpha*x + beta*y` *alpha***B* + *beta***C*

Table "Output Matrices for Triangular Solvers" shows correspondence between the output matrices and values of the parameter `matdescra` for the sparse matrix *A* for triangular solvers.

Output Matrices for Triangular Solvers

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
T	L	N	`alpha*inv(op(L))*x` *alpha**inv(op(*L*))**B*
T	L	U	`alpha*inv(op(L))*x` *alpha**inv(op(*L*))**B*
T	U	N	`alpha*inv(op(U))*x` *alpha**inv(op(*U*))**B*
T	U	U	`alpha*inv(op(U))*x` *alpha**inv(op(*U*))**B*
D	ignored	N	`alpha*inv(D)*x` *alpha**inv(*D*)**B*
D	ignored	U	`alpha*x` *alpha***B*

Sparse BLAS Level 2 and Level 3 Routines.

> **NOTE** The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines are deprecated. Use the corresponding routine from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface as indicated in the description for each routine.

Table "Sparse BLAS Level 2 and Level 3 Routines" lists the sparse BLAS Level 2 and Level 3 routines described in more detail later in this section.

Sparse BLAS Level 2 and Level 3 Routines

Routine/Function	Description

Routine/Function	Description
Simplified interface, one-based indexing	
mkl_?csrgemv	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation)
mkl_?bsrgemv	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation).
mkl_?coogemv	Computes matrix - vector product of a sparse general matrix in the coordinate format.
mkl_?diagemv	Computes matrix - vector product of a sparse general matrix in the diagonal format.
mkl_?csrsymv	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation)
mkl_?bsrsymv	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation).
mkl_?coosymv	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format.
mkl_?diasymv	Computes matrix - vector product of a sparse symmetrical matrix in the diagonal format.
mkl_?csrtrsv	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation).
mkl_?bsrtrsv	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation).
mkl_?cootrsv	Triangular solvers with simplified interface for a sparse matrix in the coordinate format.
mkl_?diatrsv	Triangular solvers with simplified interface for a sparse matrix in the diagonal format.
Simplified interface, zero-based indexing	
mkl_cspblas_?csrgemv	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation) with zero-based indexing.
mkl_cspblas_?bsrgemv	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation)with zero-based indexing.
mkl_cspblas_?coogemv	Computes matrix - vector product of a sparse general matrix in the coordinate format with zero-based indexing.
mkl_cspblas_?csrsymv	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation) with zero-based indexing
mkl_cspblas_?bsrsymv	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation) with zero-based indexing.

Routine/Function	Description
mkl_cspblas_?coosymv	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format with zero-based indexing.
mkl_cspblas_?csrtrsv	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing.
mkl_cspblas_?bsrtrsv	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation) with zero-based indexing.
mkl_cspblas_?cootrsv	Triangular solver with simplified interface for a sparse matrix in the coordinate format with zero-based indexing.
Typical (conventional) interface, one-based and zero-based indexing	
mkl_?csrmv	Computes matrix - vector product of a sparse matrix in the CSR format.
mkl_?bsrmv	Computes matrix - vector product of a sparse matrix in the BSR format.
mkl_?cscmv	Computes matrix - vector product for a sparse matrix in the CSC format.
mkl_?coomv	Computes matrix - vector product for a sparse matrix in the coordinate format.
mkl_?csrsv	Solves a system of linear equations for a sparse matrix in the CSR format.
mkl_?bsrsv	Solves a system of linear equations for a sparse matrix in the BSR format.
mkl_?cscsv	Solves a system of linear equations for a sparse matrix in the CSC format.
mkl_?coosv	Solves a system of linear equations for a sparse matrix in the coordinate format.
mkl_?csrmm	Computes matrix - matrix product of a sparse matrix in the CSR format
mkl_?bsrmm	Computes matrix - matrix product of a sparse matrix in the BSR format.
mkl_?cscmm	Computes matrix - matrix product of a sparse matrix in the CSC format
mkl_?coomm	Computes matrix - matrix product of a sparse matrix in the coordinate format.
mkl_?csrsm	Solves a system of linear matrix equations for a sparse matrix in the CSR format.
mkl_?bsrsm	Solves a system of linear matrix equations for a sparse matrix in the BSR format.
mkl_?cscsm	Solves a system of linear matrix equations for a sparse matrix in the CSC format.

Routine/Function	Description
mkl_?coosm	Solves a system of linear matrix equations for a sparse matrix in the coordinate format.
Typical (conventional) interface, one-based indexing	
mkl_?diamv	Computes matrix - vector product of a sparse matrix in the diagonal format.
mkl_?skymv	Computes matrix - vector product for a sparse matrix in the skyline storage format.
mkl_?diasv	Solves a system of linear equations for a sparse matrix in the diagonal format.
mkl_?skysv	Solves a system of linear equations for a sparse matrix in the skyline format.
mkl_?diamm	Computes matrix - matrix product of a sparse matrix in the diagonal format.
mkl_?skymm	Computes matrix - matrix product of a sparse matrix in the skyline storage format.
mkl_?diasm	Solves a system of linear matrix equations for a sparse matrix in the diagonal format.
mkl_?skysm	Solves a system of linear matrix equations for a sparse matrix in the skyline storage format.
Auxiliary routines	
Matrix converters	
mkl_?dnscsr	Converts a sparse matrix in uncompressed representation to CSR format (3-array variation) and vice versa.
mkl_?csrcoo	Converts a sparse matrix in CSR format (3-array variation) to coordinate format and vice versa.
mkl_?csrbsr	Converts a sparse matrix in CSR format to BSR format (3-array variations) and vice versa.
mkl_?csrcsc	Converts a sparse matrix in CSR format to CSC format and vice versa (3-array variations).
mkl_?csrdia	Converts a sparse matrix in CSR format (3-array variation) to diagonal format and vice versa.
mkl_?csrsky	Converts a sparse matrix in CSR format (3-array variation) to sky line format and vice versa.
Operations on sparse matrices	
mkl_?csradd	Computes the sum of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.
mkl_?csrmultcsr	Computes the product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.

Routine/Function	Description
mkl_?csrmultd	Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix.

mkl_?csrgemv

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
call mkl_scsrgemv(transa, m, a, ia, ja, x, y)
call mkl_dcsrgemv(transa, m, a, ia, ja, x, y)
call mkl_ccsrgemv(transa, m, a, ia, ja, x, y)
call mkl_zcsrgemv(transa, m, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?csrgemv routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A^T*x,
```

where:

x and *y* are vectors,

A is an *m*-by-*m* sparse square matrix in the CSR format (3-array variation), A^T is the transpose of *A*.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation. If *transa* = 'N' or 'n', then as *y* := *A*x* If *transa* = 'T' or 't' or 'C' or 'c', then *y* := A^T*x,
m	INTEGER. Number of rows of the matrix *A*.

a

REAL for mkl_scsrgemv.

DOUBLE PRECISION for mkl_dcsrgemv.

COMPLEX for mkl_ccsrgemv.

DOUBLE COMPLEX for mkl_zcsrgemv.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ia

INTEGER. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

ja

INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

x

REAL for mkl_scsrgemv.

DOUBLE PRECISION for mkl_dcsrgemv.

COMPLEX for mkl_ccsrgemv.

DOUBLE COMPLEX for mkl_zcsrgemv.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL for mkl_scsrgemv.

DOUBLE PRECISION for mkl_dcsrgemv.

COMPLEX for mkl_ccsrgemv.

DOUBLE COMPLEX for mkl_zcsrgemv.

Array, size at least *m*.

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrgemv(transa, m, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL       a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcsrgemv(transa, m, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION       a(*), x(*), y(*)

SUBROUTINE mkl_ccsrgemv(transa, m, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m
   INTEGER      ia(*), ja(*)
   COMPLEX       a(*), x(*), y(*)

SUBROUTINE mkl_zcsrgemv(transa, m, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER           m
   INTEGER           ia(*), ja(*)
   DOUBLE COMPLEX    a(*), x(*), y(*)
```

mkl_?bsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
call mkl_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrgemv routine performs a matrix-vector operation defined as

$y := A*x$

or

$y := A^{T}*x$,

where:

x and *y* are vectors,

A is an *m*-by-*m* block sparse square matrix in the BSR format (3-array variation), A^T is the transpose of *A*.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then the matrix-vector product is computed as *y* := *A***x*

If *transa* = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as *y* := A^T**x*,

m

INTEGER. Number of block rows of the matrix *A*.

lb

INTEGER. Size of the block in the matrix *A*.

a

REAL for mkl_sbsrgemv.

DOUBLE PRECISION for mkl_dbsrgemv.

COMPLEX for mkl_cbsrgemv.

DOUBLE COMPLEX for mkl_zbsrgemv.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb***lb*. Refer to *values* array description in BSR Format for more details.

ia

INTEGER. Array of length (*m* + 1), containing indices of block in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in BSR Format for more details.

ja

INTEGER. Array containing the column indices for each non-zero block in the matrix *A*.

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in BSR Format for more details.

x

REAL for mkl_sbsrgemv.

DOUBLE PRECISION for mkl_dbsrgemv.

COMPLEX for mkl_cbsrgemv.

DOUBLE COMPLEX for mkl_zbsrgemv.

Array, size (*m***lb*).

On entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL for mkl_sbsrgemv. DOUBLE PRECISION for mkl_dbsrgemv. COMPLEX for mkl_cbsrgemv. DOUBLE COMPLEX for mkl_zbsrgemv. Array, size at least (*m***lb*). On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   REAL          a(*), x(*), y(*)

SUBROUTINE mkl_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION     a(*), x(*), y(*)

SUBROUTINE mkl_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   COMPLEX       a(*), x(*), y(*)

SUBROUTINE mkl_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX       a(*), x(*), y(*)
```

mkl_?coogemv

Computes matrix-vector product of a sparse general matrix stored in the coordinate format with one-based indexing (deprecated).

Syntax

```
call mkl_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?coogemv routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A^T*x,
```

where:

x and *y* are vectors,

A is an *m*-by-*m* sparse square matrix in the coordinate format, A^T is the transpose of *A*.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation. If *transa* = 'N' or 'n', then the matrix-vector product is computed as *y* := *A***x* If *transa* = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as *y* := A^T**x*,
m	INTEGER. Number of rows of the matrix *A*.
val	REAL for mkl_scoogemv. DOUBLE PRECISION for mkl_dcoogemv. COMPLEX for mkl_ccoogemv. DOUBLE COMPLEX for mkl_zcoogemv. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.

Refer to *values* array description in Coordinate Format for more details.

rowind INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in Coordinate Format for more details.

colind INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.

nnz INTEGER. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in Coordinate Format for more details.

x REAL for mkl_scoogemv.

DOUBLE PRECISION for mkl_dcoogemv.

COMPLEX for mkl_ccoogemv.

DOUBLE COMPLEX for mkl_zcoogemv.

Array, size is *m*.

One entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for mkl_scoogemv.

DOUBLE PRECISION for mkl_dcoogemv.

COMPLEX for mkl_ccoogemv.

DOUBLE COMPLEX for mkl_zcoogemv.

Array, size at least *m*.

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  REAL            val(*), x(*), y(*)

SUBROUTINE mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION      val(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX     val(*), x(*), y(*)

SUBROUTINE mkl_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX        val(*), x(*), y(*)
```

mkl_?diagemv

Computes matrix - vector product of a sparse general matrix stored in the diagonal format with one-based indexing (deprecated).

Syntax

```
call mkl_sdiagemv(transa, m, val, lval, idiag, ndiag, x, y)
call mkl_ddiagemv(transa, m, val, lval, idiag, ndiag, x, y)
call mkl_cdiagemv(transa, m, val, lval, idiag, ndiag, x, y)
call mkl_zdiagemv(transa, m, val, lval, idiag, ndiag, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?diagemv routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A^T*x,
```

where:

x and *y* are vectors,

A is an *m*-by-*m* sparse square matrix in the diagonal storage format, A^T is the transpose of *A*.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then $y := A*x$

If *transa* = 'T' or 't' or 'C' or 'c', then $y := A^{T}*x$,

m

INTEGER. Number of rows of the matrix *A*.

val

REAL for mkl_sdiagemv.

DOUBLE PRECISION for mkl_ddiagemv.

COMPLEX for mkl_ccsrgemv.

DOUBLE COMPLEX for mkl_zdiagemv.

Two-dimensional array of size *lval***ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.

lval

INTEGER. Leading dimension of *val*$lval \geq m$. Refer to *lval* description in Diagonal Storage Scheme for more details.

idiag

INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*.

Refer to *distance* array description in Diagonal Storage Scheme for more details.

ndiag

INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.

x

REAL for mkl_sdiagemv.

DOUBLE PRECISION for mkl_ddiagemv.

COMPLEX for mkl_ccsrgemv.

DOUBLE COMPLEX for mkl_zdiagemv.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL for mkl_sdiagemv.

DOUBLE PRECISION for mkl_ddiagemv.

COMPLEX for mkl_ccsrgemv.

DOUBLE COMPLEX for mkl_zdiagemv.

Array, size at least *m*.

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiagemv(transa, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   transa
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  REAL           val(lval,*), x(*), y(*)

SUBROUTINE mkl_ddiagemv(transa, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   transa
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE PRECISION       val(lval,*), x(*), y(*)

SUBROUTINE mkl_cdiagemv(transa, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   transa
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  COMPLEX    val(lval,*), x(*), y(*)

SUBROUTINE mkl_zdiagemv(transa, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   transa
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE COMPLEX        val(lval,*), x(*), y(*)
```

mkl_?csrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
call mkl_scsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_dcsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_ccsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_zcsrsymv(uplo, m, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrsymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation).

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo

`CHARACTER*1`. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = `'U'` or `'u'`, then the upper triangle of the matrix *A* is used.

If *uplo* = `'L'` or `'l'`, then the low triangle of the matrix *A* is used.

m

`INTEGER`. Number of rows of the matrix *A*.

a

`REAL` for `mkl_scsrsymv`.

`DOUBLE PRECISION` for `mkl_dcsrsymv`.

`COMPLEX` for `mkl_ccsrsymv`.

`DOUBLE COMPLEX` for `mkl_zcsrsymv`.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ia

`INTEGER`. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

ja

`INTEGER`. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

x

`REAL` for `mkl_scsrsymv`.

DOUBLE PRECISION for mkl_dcsrsymv.

COMPLEX for mkl_ccsrsymv.

DOUBLE COMPLEX for mkl_zcsrsymv.

Array, size is m.

On entry, the array x must contain the vector x.

Output Parameters

y

REAL for mkl_scsrsymv.

DOUBLE PRECISION for mkl_dcsrsymv.

COMPLEX for mkl_ccsrsymv.

DOUBLE COMPLEX for mkl_zcsrsymv.

Array, size at least m.

On exit, the array y must contain the vector y.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL           a(*), x(*), y(*)

SUBROUTINE mkl_dcsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION       a(*), x(*), y(*)

SUBROUTINE mkl_ccsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m
   INTEGER      ia(*), ja(*)
   COMPLEX       a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX       a(*), x(*), y(*)
```

mkl_?bsrsymv

Computes matrix-vector product of a sparse symmetrical matrix stored in the BSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
call mkl_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrsymv routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation).

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo — CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is considered.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

m INTEGER. Number of block rows of the matrix *A*.

lb INTEGER. Size of the block in the matrix *A*.

a REAL for mkl_sbsrsymv.

DOUBLE PRECISION for mkl_dbsrsymv.

COMPLEX for mkl_cbsrsymv.

DOUBLE COMPLEX for mkl_zcsrgemv.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb***lb*. Refer to *values* array description in BSR Format for more details.

ia INTEGER. Array of length (*m* + 1), containing indices of block in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in BSR Format for more details.

ja INTEGER. Array containing the column indices for each non-zero block in the matrix *A*.

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in BSR Format for more details.

x REAL for mkl_sbsrsymv.

DOUBLE PRECISION for mkl_dbsrsymv.

COMPLEX for mkl_cbsrsymv.

DOUBLE COMPLEX for mkl_zcsrgemv.

Array, size (*m***lb*).

On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for mkl_sbsrsymv.

DOUBLE PRECISION for mkl_dbsrsymv.

COMPLEX for mkl_cbsrsymv.

DOUBLE COMPLEX for mkl_zcsrgemv.

Array, size at least (*m***lb*).

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   REAL          a(*), x(*), y(*)

SUBROUTINE mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION       a(*), x(*), y(*)

SUBROUTINE mkl_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   COMPLEX    a(*), x(*), y(*)

SUBROUTINE mkl_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX       a(*), x(*), y(*)
```

mkl_?coosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with one-based indexing (deprecated).

Syntax

```
call mkl_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_ccoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?coosymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo

`CHARACTER*1`. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = `'U'` or `'u'`, then the upper triangle of the matrix *A* is used.

If *uplo* = `'L'` or `'l'`, then the low triangle of the matrix *A* is used.

m

`INTEGER`. Number of rows of the matrix *A*.

val

`REAL` for `mkl_scoosymv`.

`DOUBLE PRECISION` for `mkl_dcoosymv`.

`COMPLEX` for `mkl_ccoosymv`.

`DOUBLE COMPLEX` for `mkl_zcoosymv`.

Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.

Refer to *values* array description in Coordinate Format for more details.

rowind

`INTEGER`. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in Coordinate Format for more details.

colind

`INTEGER`. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.

nnz

`INTEGER`. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in Coordinate Format for more details.

x

`REAL` for `mkl_scoosymv`.

`DOUBLE PRECISION` for `mkl_dcoosymv`.

COMPLEX for mkl_ccoosymv.

DOUBLE COMPLEX for mkl_zcoosymv.

Array, size is m.

On entry, the array x must contain the vector x.

Output Parameters

y

REAL for mkl_scoosymv.

DOUBLE PRECISION for mkl_dcoosymv.

COMPLEX for mkl_ccoosymv.

DOUBLE COMPLEX for mkl_zcoosymv.

Array, size at least m.

On exit, the array y must contain the vector y.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  REAL           val(*), x(*), y(*)

SUBROUTINE mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION     val(*), x(*), y(*)

SUBROUTINE mkl_cdcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX     val(*), x(*), y(*)

SUBROUTINE mkl_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX        val(*), x(*), y(*)
```

mkl_?diasymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the diagonal format with one-based indexing (deprecated).

Syntax

```
call mkl_sdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
call mkl_ddiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
call mkl_cdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
call mkl_zdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?diasymv routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is used. If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used. If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.
m	INTEGER. Number of rows of the matrix *A*.
val	REAL for mkl_sdiasymv. DOUBLE PRECISION for mkl_ddiasymv. COMPLEX for mkl_cdiasymv. DOUBLE COMPLEX for mkl_zdiasymv.

	Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.
lval	INTEGER. Leading dimension of *val*, *lval*≥*m*. Refer to *lval* description in Diagonal Storage Scheme for more details.
idiag	INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*. Refer to *distance* array description in Diagonal Storage Scheme for more details.
ndiag	INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.
x	REAL for mkl_sdiasymv. DOUBLE PRECISION for mkl_ddiasymv. COMPLEX for mkl_cdiasymv. DOUBLE COMPLEX for mkl_zdiasymv. Array, size is *m*. On entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL for mkl_sdiasymv. DOUBLE PRECISION for mkl_ddiasymv. COMPLEX for mkl_cdiasymv. DOUBLE COMPLEX for mkl_zdiasymv. Array, size at least *m*. On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   uplo
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  REAL           val(lval,*), x(*), y(*)

SUBROUTINE mkl_ddiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   uplo
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE PRECISION      val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   uplo
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  COMPLEX    val(lval,*), x(*), y(*)

SUBROUTINE mkl_zdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   uplo
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE COMPLEX        val(lval,*), x(*), y(*)
```

mkl_?csrtrsv

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
call mkl_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?csrtrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3 array variation):

$A*y = x$

or

$A^T*y = x$,

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo

CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

transa

CHARACTER*1. Specifies the system of linear equations.

If *transa* = 'N' or 'n', then $A*y = x$

If *transa* = 'T' or 't' or 'C' or 'c', then $A^T*y = x$,

diag

CHARACTER*1. Specifies whether *A* is unit triangular.

If *diag* = 'U' or 'u', then *A* is a unit triangular.

If *diag* = 'N' or 'n', then *A* is not unit triangular.

m

INTEGER. Number of rows of the matrix *A*.

a

REAL for mkl_scsrtrmv.

DOUBLE PRECISION for mkl_dcsrtrmv.

COMPLEX for mkl_ccsrtrmv.

DOUBLE COMPLEX for mkl_zcsrtrmv.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

ia

INTEGER. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

ja

INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

NOTE
Column indices must be sorted in increasing order for each row.

x REAL for mkl_scsrtrmv.
DOUBLE PRECISION for mkl_dcsrtrmv.
COMPLEX for mkl_ccsrtrmv.
DOUBLE COMPLEX for mkl_zcsrtrmv.
Array, size is *m*.
On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for mkl_scsrtrmv.
DOUBLE PRECISION for mkl_dcsrtrmv.
COMPLEX for mkl_ccsrtrmv.
DOUBLE COMPLEX for mkl_zcsrtrmv.
Array, size at least *m*.
Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL          a(*), x(*), y(*)

SUBROUTINE mkl_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION     a(*), x(*), y(*)

SUBROUTINE mkl_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m
   INTEGER      ia(*), ja(*)
   COMPLEX     a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX      a(*), x(*), y(*)
```

mkl_?bsrtrsv

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
call mkl_sbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrtrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) :

```
y := A*x
```

or

```
y := A^T*x,
```

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo CHARACTER*1. Specifies the upper or low triangle of the matrix *A* is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

transa CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then the matrix-vector product is computed as *y* := *A***x*

If *transa* = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as *y* := A^T**x*.

diag CHARACTER*1. Specifies whether *A* is a unit triangular matrix.

If *diag* = 'U' or 'u', then *A* is a unit triangular.

If *diag* = 'N' or 'n', then *A* is not a unit triangular.

m INTEGER. Number of block rows of the matrix *A*.

lb INTEGER. Size of the block in the matrix *A*.

a REAL for mkl_sbsrtrsv.

DOUBLE PRECISION for mkl_dbsrtrsv.

COMPLEX for mkl_cbsrtrsv.

DOUBLE COMPLEX for mkl_zbsrtrsv.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb***lb*. Refer to *values* array description in BSR Format for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

ia INTEGER. Array of length (*m* + 1), containing indices of block in the array *a*, such that *ia*(*I*) is the index in the array *a* of the first non-zero element from the row *I*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in BSR Format for more details.

ja INTEGER.

Array containing the column indices for each non-zero block in the matrix *A*.

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in BSR Format for more details.

x REAL for mkl_sbsrtrsv.

DOUBLE PRECISION for mkl_dbsrtrsv.

COMPLEX for mkl_cbsrtrsv.

DOUBLE COMPLEX for mkl_zbsrtrsv.

Array, size (*m***lb*).

On entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL for mkl_sbsrtrsv.

DOUBLE PRECISION for mkl_dbsrtrsv.

COMPLEX for mkl_cbsrtrsv.

DOUBLE COMPLEX for mkl_zbsrtrsv.

Array, size at least (*m***lb*).

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   REAL          a(*), x(*), y(*)

SUBROUTINE mkl_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION       a(*), x(*), y(*)

SUBROUTINE mkl_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   COMPLEX       a(*), x(*), y(*)

SUBROUTINE mkl_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX       a(*), x(*), y(*)
```

mkl_?cootrsv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with one-based indexing (deprecated).

Syntax

```
call mkl_scootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_ccootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_zcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?cootrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format:

$A*y = x$

or

$A^T*y = x$,

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo — CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is considered.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

transa — CHARACTER*1. Specifies the system of linear equations.

If *transa* = 'N' or 'n', then $A*y = x$

If *transa* = 'T' or 't' or 'C' or 'c', then $A^T*y = x$,

diag — CHARACTER*1. Specifies whether *A* is unit triangular.

If *diag* = 'U' or 'u', then *A* is unit triangular.

If *diag* = 'N' or 'n', then *A* is not unit triangular.

m INTEGER. Number of rows of the matrix *A*.

val REAL for mkl_scootrsv.

DOUBLE PRECISION for mkl_dcootrsv.

COMPLEX for mkl_ccootrsv.

DOUBLE COMPLEX for mkl_zcootrsv.

Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.

Refer to *values* array description in Coordinate Format for more details.

rowind INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in Coordinate Format for more details.

colind INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.

nnz INTEGER. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in Coordinate Format for more details.

x REAL for mkl_scootrsv.

DOUBLE PRECISION for mkl_dcootrsv.

COMPLEX for mkl_ccootrsv.

DOUBLE COMPLEX for mkl_zcootrsv.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for mkl_scootrsv.

DOUBLE PRECISION for mkl_dcootrsv.

COMPLEX for mkl_ccootrsv.

DOUBLE COMPLEX for mkl_zcootrsv.

Array, size at least *m*.

Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  REAL            val(*), x(*), y(*)

SUBROUTINE mkl_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION        val(*), x(*), y(*)

SUBROUTINE mkl_ccootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX        val(*), x(*), y(*)

SUBROUTINE mkl_zcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX        val(*), x(*), y(*)
```

mkl_?diatrsv

Triangular solvers with simplified interface for a sparse matrix in the diagonal format with one-based indexing (deprecated).

Syntax

```
call mkl_sdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
call mkl_ddiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
call mkl_cdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
call mkl_zdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?diatrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

$A*y = x$

or

$A^T*y = x$,

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo

`CHARACTER*1`. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = `'U'` or `'u'`, then the upper triangle of the matrix *A* is used.

If *uplo* = `'L'` or `'l'`, then the low triangle of the matrix *A* is used.

transa

`CHARACTER*1`. Specifies the system of linear equations.

If *transa* = `'N'` or `'n'`, then $A*y = x$

If *transa* = `'T'` or `'t'` or `'C'` or `'c'`, then $A^T*y = x$,

diag

`CHARACTER*1`. Specifies whether *A* is unit triangular.

If *diag* = `'U'` or `'u'`, then *A* is unit triangular.

If *diag* = `'N'` or `'n'`, then *A* is not unit triangular.

m

`INTEGER`. Number of rows of the matrix *A*.

val

`REAL` for `mkl_sdiatrsv`.

`DOUBLE PRECISION` for `mkl_ddiatrsv`.

`COMPLEX` for `mkl_cdiatrsv`.

`DOUBLE COMPLEX` for `mkl_zdiatrsv`.

Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.

lval INTEGER. Leading dimension of *val*, *lval*≥*m*. Refer to *lval* description in Diagonal Storage Scheme for more details.

idiag INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*.

NOTE
All elements of this array must be sorted in increasing order.

Refer to *distance* array description in Diagonal Storage Scheme for more details.

ndiag INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.

x REAL for mkl_sdiatrsv.

DOUBLE PRECISION for mkl_ddiatrsv.

COMPLEX for mkl_cdiatrsv.

DOUBLE COMPLEX for mkl_zdiatrsv.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for mkl_sdiatrsv.

DOUBLE PRECISION for mkl_ddiatrsv.

COMPLEX for mkl_cdiatrsv.

DOUBLE COMPLEX for mkl_zdiatrsv.

Array, size at least *m*.

Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, lval, ndiag
  INTEGER       indiag(*)
  REAL            val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_ddiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, lval, ndiag
  INTEGER       indiag(*)
  DOUBLE PRECISION       val(lval,*), x(*), y(*)

SUBROUTINE mkl_cdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, lval, ndiag
  INTEGER       indiag(*)
  COMPLEX     val(lval,*), x(*), y(*)

SUBROUTINE mkl_zdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, lval, ndiag
  INTEGER       indiag(*)
  DOUBLE COMPLEX        val(lval,*), x(*), y(*)
```

mkl_cspblas_?csrgemv

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

```
call mkl_cspblas_scsrgemv(transa, m, a, ia, ja, x, y)
call mkl_cspblas_dcsrgemv(transa, m, a, ia, ja, x, y)
call mkl_cspblas_ccsrgemv(transa, m, a, ia, ja, x, y)
call mkl_cspblas_zcsrgemv(transa, m, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?csrgemv routine performs a matrix-vector operation defined as

$y := A*x$

or

$y := A^{T}*x$,

where:

x and *y* are vectors,

A is an *m*-by-*m* sparse square matrix in the CSR format (3-array variation) with zero-based indexing, A^T is the transpose of *A*.

NOTE
This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

`CHARACTER*1`. Specifies the operation.

If *transa* = `'N'` or `'n'`, then the matrix-vector product is computed as *y* := *A***x*

If *transa* = `'T'` or `'t'` or `'C'` or `'c'`, then the matrix-vector product is computed as *y* := A^T**x*,

m

`INTEGER`. Number of rows of the matrix *A*.

a

`REAL` for `mkl_cspblas_scsrgemv`.

`DOUBLE PRECISION` for `mkl_cspblas_dcsrgemv`.

`COMPLEX` for `mkl_cspblas_ccsrgemv`.

`DOUBLE COMPLEX` for `mkl_cspblas_zcsrgemv`.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ia

`INTEGER`. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(*I*) is the index in the array *a* of the first non-zero element from the row *I*. The value of the last element *ia*(*m*) is equal to the number of non-zeros. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

ja

`INTEGER`. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

x

`REAL` for `mkl_cspblas_scsrgemv`.

`DOUBLE PRECISION` for `mkl_cspblas_dcsrgemv`.

`COMPLEX` for `mkl_cspblas_ccsrgemv`.

`DOUBLE COMPLEX` for `mkl_cspblas_zcsrgemv`.

Array, size is *m*.

One entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL for mkl_cspblas_scsrgemv. DOUBLE PRECISION for mkl_cspblas_dcsrgemv. COMPLEX for mkl_cspblas_ccsrgemv. DOUBLE COMPLEX for mkl_cspblas_zcsrgemv. Array, size at least *m*. On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scsrgemv(transa, m, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL           a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dcsrgemv(transa, m, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION       a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccsrgemv(transa, m, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m
   INTEGER      ia(*), ja(*)
   COMPLEX       a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcsrgemv(transa, m, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX      a(*), x(*), y(*)
```

mkl_cspblas_?bsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

```
call mkl_cspblas_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cspblas_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cspblas_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cspblas_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?bsrgemv routine performs a matrix-vector operation defined as

$y := A*x$

or

$y := A^{T}*x$,

where:

x and *y* are vectors,

A is an *m*-by-*m* block sparse square matrix in the BSR format (3-array variation) with zero-based indexing, A^{T} is the transpose of *A*.

> **NOTE**
> This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation. If *transa* = 'N' or 'n', then the matrix-vector product is computed as *y* := *A***x* If *transa* = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A^{T}*x$,
m	INTEGER. Number of block rows of the matrix *A*.
lb	INTEGER. Size of the block in the matrix *A*.
a	REAL for mkl_cspblas_sbsrgemv. DOUBLE PRECISION for mkl_cspblas_dbsrgemv. COMPLEX for mkl_cspblas_cbsrgemv. DOUBLE COMPLEX for mkl_cspblas_zbsrgemv.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by `lb*lb`. Refer to `values` array description in BSR Format for more details.

ia — `INTEGER`. Array of length `(m + 1)`, containing indices of block in the array `a`, such that `ia(i)` is the index in the array `a` of the first non-zero element from the row *i*. The value of the last element `ia(m + 1)` is equal to the number of non-zero blocks. Refer to `rowIndex` array description in BSR Format for more details.

ja — `INTEGER`. Array containing the column indices for each non-zero block in the matrix *A*.

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to `columns` array description in BSR Format for more details.

x — `REAL` for `mkl_cspblas_sbsrgemv`.

`DOUBLE PRECISION` for `mkl_cspblas_dbsrgemv`.

`COMPLEX` for `mkl_cspblas_cbsrgemv`.

`DOUBLE COMPLEX` for `mkl_cspblas_zbsrgemv`.

Array, size `(m*lb)`.

On entry, the array `x` must contain the vector *x*.

Output Parameters

y — `REAL` for `mkl_cspblas_sbsrgemv`.

`DOUBLE PRECISION` for `mkl_cspblas_dbsrgemv`.

`COMPLEX` for `mkl_cspblas_cbsrgemv`.

`DOUBLE COMPLEX` for `mkl_cspblas_zbsrgemv`.

Array, size at least `(m*lb)`.

On exit, the array `y` must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   REAL          a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION       a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   COMPLEX       a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX       a(*), x(*), y(*)
```

mkl_cspblas_?coogemv

Computes matrix - vector product of a sparse general matrix stored in the coordinate format with zero-based indexing (deprecated).

Syntax

```
call mkl_cspblas_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_dcoogemv routine performs a matrix-vector operation defined as

$y := A*x$

or

$y := A^{T}*x$,

where:

x and *y* are vectors,

A is an *m*-by-*m* sparse square matrix in the coordinate format with zero-based indexing, A^T is the transpose of *A*.

NOTE
This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

`CHARACTER*1`. Specifies the operation.

If *transa* = `'N'` or `'n'`, then the matrix-vector product is computed as *y* := *A***x*

If *transa* = `'T'` or `'t'` or `'C'` or `'c'`, then the matrix-vector product is computed as *y* := A^T**x*.

m

`INTEGER`. Number of rows of the matrix *A*.

val

`REAL` for `mkl_cspblas_scoogemv`.

`DOUBLE PRECISION` for `mkl_cspblas_dcoogemv`.

`COMPLEX` for `mkl_cspblas_ccoogemv`.

`DOUBLE COMPLEX` for `mkl_cspblas_zcoogemv`.

Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.

Refer to *values* array description in Coordinate Format for more details.

rowind

`INTEGER`. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in Coordinate Format for more details.

colind

`INTEGER`. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.

nnz

`INTEGER`. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in Coordinate Format for more details.

x

`REAL` for `mkl_cspblas_scoogemv`.

`DOUBLE PRECISION` for `mkl_cspblas_dcoogemv`.

`COMPLEX` for `mkl_cspblas_ccoogemv`.

`DOUBLE COMPLEX` for `mkl_cspblas_zcoogemv`.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL for mkl_cspblas_scoogemv. DOUBLE PRECISION for mkl_cspblas_dcoogemv. COMPLEX for mkl_cspblas_ccoogemv. DOUBLE COMPLEX for mkl_cspblas_zcoogemv. Array, size at least *m*. On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  REAL            val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION      val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX        val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX       val(*), x(*), y(*)
```

mkl_cspblas_?csrsymv

Computes matrix-vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

```
call mkl_cspblas_scsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_cspblas_dcsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_cspblas_ccsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_cspblas_zcsrsymv(uplo, m, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?csrsymv routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation) with zero-based indexing.

> **NOTE**
> This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo

CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

m

INTEGER. Number of rows of the matrix *A*.

a

REAL for mkl_cspblas_scsrsymv.

DOUBLE PRECISION for mkl_cspblas_dcsrsymv.

COMPLEX for mkl_cspblas_ccsrsymv.

DOUBLE COMPLEX for mkl_cspblas_zcsrsymv.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ia — INTEGER. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zeros. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

ja — INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

x — REAL for mkl_cspblas_scsrsymv.

DOUBLE PRECISION for mkl_cspblas_dcsrsymv.

COMPLEX for mkl_cspblas_ccsrsymv.

DOUBLE COMPLEX for mkl_cspblas_zcsrsymv.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y — REAL for mkl_cspblas_scsrsymv.

DOUBLE PRECISION for mkl_cspblas_dcsrsymv.

COMPLEX for mkl_cspblas_ccsrsymv.

DOUBLE COMPLEX for mkl_cspblas_zcsrsymv.

Array, size at least *m*.

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL         a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dcsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1        uplo
   INTEGER            m
   INTEGER            ia(*), ja(*)
   DOUBLE PRECISION   a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_ccsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m
   INTEGER      ia(*), ja(*)
   COMPLEX      a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1      uplo
   INTEGER          m
   INTEGER          ia(*), ja(*)
   DOUBLE COMPLEX   a(*), x(*), y(*)
```

mkl_cspblas_?bsrsymv

Computes matrix-vector product of a sparse symmetrical matrix stored in the BSR format (3-arrays variation) with zero-based indexing (deprecated).

Syntax

```
call mkl_cspblas_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_cspblas_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_cspblas_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_cspblas_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?bsrsymv routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation) with zero-based indexing.

> **NOTE**
> This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo

CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

m

INTEGER. Number of block rows of the matrix *A*.

lb

INTEGER. Size of the block in the matrix *A*.

a

REAL for mkl_cspblas_sbsrsymv.

DOUBLE PRECISION for mkl_cspblas_dbsrsymv.

COMPLEX for mkl_cspblas_cbsrsymv.

DOUBLE COMPLEX for mkl_cspblas_zbsrsymv.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb*lb*. Refer to *values* array description in BSR Format for more details.

ia

INTEGER. Array of length (*m* + 1), containing indices of block in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in BSR Format for more details.

ja

INTEGER. Array containing the column indices for each non-zero block in the matrix *A*.

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in BSR Format for more details.

x

REAL for mkl_cspblas_sbsrsymv.

DOUBLE PRECISION for mkl_cspblas_dbsrsymv.

COMPLEX for mkl_cspblas_cbsrsymv.

DOUBLE COMPLEX for mkl_cspblas_zbsrsymv.

Array, size (*m*lb*).

On entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL for mkl_cspblas_sbsrsymv.

DOUBLE PRECISION for mkl_cspblas_dbsrsymv.

COMPLEX for mkl_cspblas_cbsrsymv.

DOUBLE COMPLEX for mkl_cspblas_zbsrsymv.

Array, size at least `(m*lb)`.

On exit, the array `y` must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   REAL           a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION       a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   COMPLEX    a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX    a(*), x(*), y(*)
```

mkl_cspblas_?coosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with zero-based indexing (deprecated).

Syntax

```
call mkl_cspblas_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_ccoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?coosymv routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format with zero-based indexing.

NOTE
This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo

CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

m

INTEGER. Number of rows of the matrix *A*.

val

REAL for mkl_cspblas_scoosymv.

DOUBLE PRECISION for mkl_cspblas_dcoosymv.

COMPLEX for mkl_cspblas_ccoosymv.

DOUBLE COMPLEX for mkl_cspblas_zcoosymv.

Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.

Refer to *values* array description in Coordinate Format for more details.

rowind

INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in Coordinate Format for more details.

colind

INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.

nnz

INTEGER. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in Coordinate Format for more details.

x

REAL for mkl_cspblas_scoosymv.

DOUBLE PRECISION for mkl_cspblas_dcoosymv.

COMPLEX for mkl_cspblas_ccoosymv.

DOUBLE COMPLEX for mkl_cspblas_zcoosymv.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL for mkl_cspblas_scoosymv.

DOUBLE PRECISION for mkl_cspblas_dcoosymv.

COMPLEX for mkl_cspblas_ccoosymv.

DOUBLE COMPLEX for mkl_cspblas_zcoosymv.

Array, size at least *m*.

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  REAL           val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION        val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX     val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX      val(*), x(*), y(*)
```

mkl_cspblas_?csrtrsv

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

```
call mkl_cspblas_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_cspblas_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_cspblas_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_cspblas_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?csrtrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3-array variation) with zero-based indexing:

$A*y = x$

or

$A^T*y = x$,

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE
This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo — CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

transa CHARACTER*1. Specifies the system of linear equations.

If *transa* = 'N' or 'n', then $A*y = x$

If *transa* = 'T' or 't' or 'C' or 'c', then $A^{T}*y = x$,

diag CHARACTER*1. Specifies whether matrix *A* is unit triangular.

If *diag* = 'U' or 'u', then *A* is unit triangular.

If *diag* = 'N' or 'n', then *A* is not unit triangular.

m INTEGER. Number of rows of the matrix *A*.

a REAL for mkl_cspblas_scsrtrsv.

DOUBLE PRECISION for mkl_cspblas_dcsrtrsv.

COMPLEX for mkl_cspblas_ccsrtrsv.

DOUBLE COMPLEX for mkl_cspblas_zcsrtrsv.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

ia INTEGER. Array of length *m*+1, containing indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m*) is equal to the number of non-zeros. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

ja INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

NOTE
Column indices must be sorted in increasing order for each row.

x REAL for mkl_cspblas_scsrtrsv.

DOUBLE PRECISION for mkl_cspblas_dcsrtrsv.

COMPLEX for mkl_cspblas_ccsrtrsv.

DOUBLE COMPLEX for mkl_cspblas_zcsrtrsv.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL for mkl_cspblas_scsrtrsv.

DOUBLE PRECISION for mkl_cspblas_dcsrtrsv.

COMPLEX for mkl_cspblas_ccsrtrsv.

DOUBLE COMPLEX for mkl_cspblas_zcsrtrsv.

Array, size at least *m*.

Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL          a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION     a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m
   INTEGER      ia(*), ja(*)
   COMPLEX       a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX       a(*), x(*), y(*)
```

mkl_cspblas_?bsrtrsv

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

```
call mkl_cspblas_sbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_cspblas_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_cspblas_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_cspblas_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?bsrtrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing:

```
y := A*x
```

or

```
y := A^T*x,
```

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

> **NOTE**
> This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo

CHARACTER*1. Specifies the upper or low triangle of the matrix *A* is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

transa

CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then the matrix-vector product is computed as *y* := *A*x*

If *transa* = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as *y* := A^T*x.

diag CHARACTER*1. Specifies whether matrix *A* is unit triangular or not.

If *diag* = 'U' or 'u', *A* is unit triangular.

If *diag* = 'N' or 'n', *A* is not unit triangular.

m INTEGER. Number of block rows of the matrix *A*.

lb INTEGER. Size of the block in the matrix *A*.

a REAL for mkl_cspblas_sbsrtrsv.

DOUBLE PRECISION for mkl_cspblas_dbsrtrsv.

COMPLEX for mkl_cspblas_cbsrtrsv.

DOUBLE COMPLEX for mkl_cspblas_zbsrtrsv.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb*lb*. Refer to *values* array description in BSR Format for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

ia INTEGER. Array of length (*m* + 1), containing indices of block in the array *a*, such that *ia*(*I*) is the index in the array *a* of the first non-zero element from the row *I*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zero blocks. Refer to *rowIndex* array description in BSR Format for more details.

ja INTEGER. Array containing the column indices for each non-zero block in the matrix *A*.

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in BSR Format for more details.

x REAL for mkl_cspblas_sbsrtrsv.

DOUBLE PRECISION for mkl_cspblas_dbsrtrsv.

COMPLEX for mkl_cspblas_cbsrtrsv.

DOUBLE COMPLEX for mkl_cspblas_zbsrtrsv.

Array, size (*m*lb*).

On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for mkl_cspblas_sbsrtrsv.

DOUBLE PRECISION for mkl_cspblas_dbsrtrsv.

COMPLEX for mkl_cspblas_cbsrtrsv.

DOUBLE COMPLEX for mkl_cspblas_zbsrtrsv.

Array, size at least (*m***lb*).

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_sbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   REAL          a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE PRECISION       a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   COMPLEX       a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   DOUBLE COMPLEX      a(*), x(*), y(*)
```

mkl_cspblas_?cootrsv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with zero-based indexing (deprecated).

Syntax

```
call mkl_cspblas_scootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_ccootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_zcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?cootrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format with zero-based indexing:

$A*y = x$

or

$A^T*y = x$,

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE
This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo
CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is considered.
If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.
If *uplo* = 'L' or 'l', then the low triangle of the matrix *A* is used.

transa
CHARACTER*1. Specifies the system of linear equations.
If *transa* = 'N' or 'n', then $A*y = x$
If *transa* = 'T' or 't' or 'C' or 'c', then $A^T*y = x$,

diag
CHARACTER*1. Specifies whether *A* is unit triangular.
If *diag* = 'U' or 'u', then *A* is unit triangular.
If *diag* = 'N' or 'n', then *A* is not unit triangular.

m
INTEGER. Number of rows of the matrix *A*.

val
REAL for mkl_cspblas_scootrsv.
DOUBLE PRECISION for mkl_cspblas_dcootrsv.
COMPLEX for mkl_cspblas_ccootrsv.
DOUBLE COMPLEX for mkl_cspblas_zcootrsv.

Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.

Refer to *values* array description in Coordinate Format for more details.

rowind INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in Coordinate Format for more details.

colind INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.

nnz INTEGER. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in Coordinate Format for more details.

x REAL for mkl_cspblas_scootrsv.

DOUBLE PRECISION for mkl_cspblas_dcootrsv.

COMPLEX for mkl_cspblas_ccootrsv.

DOUBLE COMPLEX for mkl_cspblas_zcootrsv.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for mkl_cspblas_scootrsv.

DOUBLE PRECISION for mkl_cspblas_dcootrsv.

COMPLEX for mkl_cspblas_ccootrsv.

DOUBLE COMPLEX for mkl_cspblas_zcootrsv.

Array, size at least *m*.

Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  REAL           val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION      val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX        val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo, transa, diag
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX      val(*), x(*), y(*)
```

mkl_?csrmv

Computes matrix - vector product of a sparse matrix stored in the CSR format (deprecated).

Syntax

```
call mkl_scsrmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_dcsrmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_ccsrmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_zcsrmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?csrmv routine performs a matrix-vector operation defined as

$y := alpha*A*x + beta*y$

or

$y := alpha*A^{T}*x + beta*y,$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in the CSR format, A^T is the transpose of *A*.

NOTE
This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

`CHARACTER*1`. Specifies the operation.

If *transa* = `'N'` or `'n'`, then *y* := *alpha***A***x* + *beta***y*

If *transa* = `'T'` or `'t'` or `'C'` or `'c'`, then *y* := *alpha***A*^{T}**x* + *beta***y*,

m

`INTEGER`. Number of rows of the matrix *A*.

k

`INTEGER`. Number of columns of the matrix *A*.

alpha

`REAL` for `mkl_scsrmv`.

`DOUBLE PRECISION` for `mkl_dcsrmv`.

`COMPLEX` for `mkl_ccsrmv`.

`DOUBLE COMPLEX` for `mkl_zcsrmv`.

Specifies the scalar *alpha*.

matdescra

`CHARACTER`. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

`REAL` for `mkl_scsrmv`.

`DOUBLE PRECISION` for `mkl_dcsrmv`.

`COMPLEX` for `mkl_ccsrmv`.

`DOUBLE COMPLEX` for `mkl_zcsrmv`.

Array containing non-zero elements of the matrix *A*.

For one-based indexing its length is *pntre*`(m)` - *pntrb*`(1)`.

For zero-based indexing its length is *pntre*`(m-1)` - *pntrb*`(0)`.

Refer to *values* array description in CSR Format for more details.

indx

`INTEGER`. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *columns* array description in CSR Format for more details.

pntrb

INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntrb*(*i*) – *pntrb*(1) + 1 is the first index of row *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of row *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in CSR Format for more details.

pntre

INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of row *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(0)–1 is the last index of row *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in CSR Format for more details.

x

REAL for mkl_scsrmv.

DOUBLE PRECISION for mkl_dcsrmv.

COMPLEX for mkl_ccsrmv.

DOUBLE COMPLEX for mkl_zcsrmv.

Array, size at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta

REAL for mkl_scsrmv.

DOUBLE PRECISION for mkl_dcsrmv.

COMPLEX for mkl_ccsrmv.

DOUBLE COMPLEX for mkl_zcsrmv.

Specifies the scalar *beta*.

y

REAL for mkl_scsrmv.

DOUBLE PRECISION for mkl_dcsrmv.

COMPLEX for mkl_ccsrmv.

DOUBLE COMPLEX for mkl_zcsrmv.

Array, size at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y

Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrmv(transa, m, k, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL           alpha, beta
  REAL           val(*), x(*), y(*)

SUBROUTINE mkl_dcsrmv(transa, m, k, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), x(*), y(*)

SUBROUTINE mkl_ccsrmv(transa, m, k, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX      alpha, beta
  COMPLEX      val(*), x(*), y(*)

SUBROUTINE mkl_zcsrmv(transa, m, k, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), x(*), y(*)
```

mkl_?bsrmv

Computes matrix - vector product of a sparse matrix stored in the BSR format (deprecated).

Syntax

```
call mkl_sbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_cbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_zbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrmv routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A^T*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* block sparse matrix in the BSR format, A^T is the transpose of *A*.

NOTE
This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa — CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then the matrix-vector product is computed as *y := alpha*A*x + beta*y*

If *transa* = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as *y := alpha*A^T*x + beta*y*,

m — INTEGER. Number of block rows of the matrix *A*.

k — INTEGER. Number of block columns of the matrix *A*.

lb	INTEGER. Size of the block in the matrix *A*.
alpha	REAL for mkl_sbsrmv. DOUBLE PRECISION for mkl_dbsrmv. COMPLEX for mkl_cbsrmv. DOUBLE COMPLEX for mkl_zbsrmv. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".
val	REAL for mkl_sbsrmv. DOUBLE PRECISION for mkl_dbsrmv. COMPLEX for mkl_cbsrmv. DOUBLE COMPLEX for mkl_zbsrmv. Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb*lb*. Refer to *values* array description in BSR Format for more details.
indx	INTEGER. Array containing the column indices for each non-zero block in the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A*. Refer to *columns* array description in BSR Format for more details.
pntrb	INTEGER. Array of length *m*. For one-based indexing: this array contains row indices, such that *pntrb*(*i*) – *pntrb*(1) + 1 is the first index of block row *i* in the array *indx*. For zero-based indexing: this array contains row indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of block row *i* in the array *indx* Refer to *pointerB* array description in BSR Format for more details.
pntre	INTEGER. Array of length *m*. For one-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of block row *i* in the array *indx*. For zero-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(0) – 1 is the last index of block row *i* in the array *indx*. Refer to *pointerE* array description in BSR Format for more details.
x	REAL for mkl_sbsrmv. DOUBLE PRECISION for mkl_dbsrmv.

COMPLEX for mkl_cbsrmv.

DOUBLE COMPLEX for mkl_zbsrmv.

Array, size at least (*k***lb*) if *transa* = 'N' or 'n', and at least (*m***lb*) otherwise. On entry, the array *x* must contain the vector *x*.

beta

REAL for mkl_sbsrmv.

DOUBLE PRECISION for mkl_dbsrmv.

COMPLEX for mkl_cbsrmv.

DOUBLE COMPLEX for mkl_zbsrmv.

Specifies the scalar *beta*.

y

REAL for mkl_sbsrmv.

DOUBLE PRECISION for mkl_dbsrmv.

COMPLEX for mkl_cbsrmv.

DOUBLE COMPLEX for mkl_zbsrmv.

Array, size at least (*m***lb*) if *transa* = 'N' or 'n', and at least (*k***lb*) otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y

Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, lb
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha, beta
  REAL          val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, lb
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), x(*), y(*)

SUBROUTINE mkl_cbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, lb
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha, beta
  COMPLEX        val(*), x(*), y(*)

SUBROUTINE mkl_zbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, lb
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), x(*), y(*)
```

mkl_?cscmv

Computes matrix-vector product for a sparse matrix in the CSC format (deprecated).

Syntax

```
call mkl_scscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_ccscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_zcscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?cscmv` routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A^T*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed sparse column (CSC) format, A^T is the transpose of *A*.

> **NOTE**
> This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

`CHARACTER*1`. Specifies the operation.

If *transa* = `'N'` or `'n'`, then *y* := *alpha**A*x + *beta**y*

If *transa* = `'T'` or `'t'` or `'C'` or `'c'`, then *y* := *alpha**A^T*x + *beta**y*,

m

`INTEGER`. Number of rows of the matrix *A*.

k

`INTEGER`. Number of columns of the matrix *A*.

alpha

`REAL` for `mkl_scscmv`.

`DOUBLE PRECISION` for `mkl_dcscmv`.

`COMPLEX` for `mkl_ccscmv`.

`DOUBLE COMPLEX` for `mkl_zcscmv`.

Specifies the scalar *alpha*.

matdescra

`CHARACTER`. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

`REAL` for `mkl_scscmv`.

`DOUBLE PRECISION` for `mkl_dcscmv`.

`COMPLEX` for `mkl_ccscmv`.

DOUBLE COMPLEX for mkl_zcscmv.

Array containing non-zero elements of the matrix *A*.

For one-based indexing its length is pntre(*k*) – *pntrb*(1).

For zero-based indexing its length is *pntre*(*m*–1) – *pntrb*(0).

Refer to *values* array description in CSC Format for more details.

indx INTEGER. Array containing the row indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *rows* array description in CSC Format for more details.

pntrb INTEGER. Array of length *k*.

For one-based indexing this array contains column indices, such that *pntrb*(*i*) – *pntrb*(1) + 1 is the first index of column *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of column *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in CSC Format for more details.

pntre INTEGER. Array of length *k*.

For one-based indexing this array contains column indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of column *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that *pntre*(*i*) – *pntrb*(1) – 1 is the last index of column *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in CSC Format for more details.

x REAL for mkl_scscmv.

DOUBLE PRECISION for mkl_dcscmv.

COMPLEX for mkl_ccscmv.

DOUBLE COMPLEX for mkl_zcscmv.

Array, size at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta REAL for mkl_scscmv.

DOUBLE PRECISION for mkl_dcscmv.

COMPLEX for mkl_ccscmv.

DOUBLE COMPLEX for mkl_zcscmv.

Specifies the scalar *beta*.

y REAL for mkl_scscmv.

DOUBLE PRECISION for mkl_dcscmv.

COMPLEX for mkl_ccscmv.

DOUBLE COMPLEX for mkl_zcscmv.

Array, size at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y — Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscmv(transa, m, k, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha, beta
  REAL          val(*), x(*), y(*)

SUBROUTINE mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), x(*), y(*)

SUBROUTINE mkl_ccscmv(transa, m, k, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha, beta
  COMPLEX        val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcscmv(transa, m, k, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), x(*), y(*)
```

mkl_?coomv

Computes matrix - vector product for a sparse matrix in the coordinate format (deprecated).

Syntax

```
call mkl_scoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
call mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
call mkl_ccoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
call mkl_zcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?coomv routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A^T*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed coordinate format, A^T is the transpose of *A*.

NOTE
This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation. If *transa* = 'N' or 'n', then $y := alpha*A*x + beta*y$ If *transa* = 'T' or 't' or 'C' or 'c', then $y := alpha*A^{T}*x + beta*y$,
m	INTEGER. Number of rows of the matrix *A*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL for mkl_scoomv. DOUBLE PRECISION for mkl_dcoomv. COMPLEX for mkl_ccoomv. DOUBLE COMPLEX for mkl_zcoomv. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".
val	REAL for mkl_scoomv. DOUBLE PRECISION for mkl_dcoomv. COMPLEX for mkl_ccoomv. DOUBLE COMPLEX for mkl_zcoomv. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order. Refer to *values* array description in Coordinate Format for more details.
rowind	INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*. Refer to *rows* array description in Coordinate Format for more details.
colind	INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.
nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
x	REAL for mkl_scoomv. DOUBLE PRECISION for mkl_dcoomv. COMPLEX for mkl_ccoomv.

DOUBLE COMPLEX for mkl_zcoomv.

Array, size at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta

REAL for mkl_scoomv.

DOUBLE PRECISION for mkl_dcoomv.

COMPLEX for mkl_ccoomv.

DOUBLE COMPLEX for mkl_zcoomv.

Specifies the scalar *beta*.

y

REAL for mkl_scoomv.

DOUBLE PRECISION for mkl_dcoomv.

COMPLEX for mkl_ccoomv.

DOUBLE COMPLEX for mkl_zcoomv.

Array, size at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y

Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, nnz
  INTEGER       rowind(*), colind(*)
  REAL          alpha, beta
  REAL          val(*), x(*), y(*)

SUBROUTINE mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX        alpha, beta
  COMPLEX        val(*), x(*), y(*)

SUBROUTINE mkl_zcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), x(*), y(*)
```

mkl_?csrsv

Solves a system of linear equations for a sparse matrix in the CSR format (deprecated).

Syntax

```
call mkl_scsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_dcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_ccsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_zcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?csrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSR format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A^T)*x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE
This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

`CHARACTER*1`. Specifies the system of linear equations.

If *transa* = `'N'` or `'n'`, then *y* := *alpha**inv(*A*)**x*

If *transa* = `'T'` or `'t'` or `'C'` or `'c'`, then *y* := *alpha**inv(A^T)**x*,

m

`INTEGER`. Number of columns of the matrix *A*.

alpha

`REAL` for `mkl_scsrsv`.

`DOUBLE PRECISION` for `mkl_dcsrsv`.

`COMPLEX` for `mkl_ccsrsv`.

`DOUBLE COMPLEX` for `mkl_zcsrsv`.

Specifies the scalar *alpha*.

matdescra

`CHARACTER`. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

`REAL` for `mkl_scsrsv`.

`DOUBLE PRECISION` for `mkl_dcsrsv`.

`COMPLEX` for `mkl_ccsrsv`.

`DOUBLE COMPLEX` for `mkl_zcsrsv`.

Array containing non-zero elements of the matrix *A*.

For one-based indexing its length is *pntre*(m) – *pntrb*(1).

For zero-based indexing its length is *pntre*(m-1) – *pntrb*(0).

Refer to *values* array description in CSR Format for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx

`INTEGER`. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *columns* array description in CSR Format for more details.

NOTE
Column indices must be sorted in increasing order for each row.

pntrb

INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntrb*(*i*) – *pntrb*(1) + 1 is the first index of row *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of row *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in CSR Format for more details.

pntre

INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of row *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(0) – 1 is the last index of row *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in CSR Format for more details.

x

REAL for mkl_scsrsv.

DOUBLE PRECISION for mkl_dcsrsv.

COMPLEX for mkl_ccsrsv.

DOUBLE COMPLEX for mkl_zcsrsv.

Array, size at least *m*.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

REAL for mkl_scsrsv.

DOUBLE PRECISION for mkl_dcsrsv.

COMPLEX for mkl_ccsrsv.

DOUBLE COMPLEX for mkl_zcsrsv.

Array, size at least *m*.

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y

Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha
  REAL          val(*)
  REAL          x(*), y(*)

SUBROUTINE mkl_dcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*)
  DOUBLE PRECISION        x(*), y(*)

SUBROUTINE mkl_ccsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha
  COMPLEX        val(*)
  COMPLEX        x(*), y(*)

SUBROUTINE mkl_zcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*)
  DOUBLE COMPLEX        x(*), y(*)
```

mkl_?bsrsv

Solves a system of linear equations for a sparse matrix in the BSR format (deprecated).

Syntax

```
call mkl_sbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_dbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_cbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_zbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the BSR format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A^T)* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

> **NOTE**
> This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation. If *transa* = 'N' or 'n', then *y* := *alpha**inv(*A*)**x* If *transa* = 'T' or 't' or 'C' or 'c', then *y* := *alpha**inv(A^T)* *x*,
m	INTEGER. Number of block columns of the matrix *A*.
lb	INTEGER. Size of the block in the matrix *A*.
alpha	REAL for mkl_sbsrsv. DOUBLE PRECISION for mkl_dbsrsv.

`COMPLEX` for `mkl_cbsrsv`.

`DOUBLE COMPLEX` for `mkl_zbsrsv`.

Specifies the scalar *alpha*.

matdescra

`CHARACTER`. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

`REAL` for `mkl_sbsrsv`.

`DOUBLE PRECISION` for `mkl_dbsrsv`.

`COMPLEX` for `mkl_cbsrsv`.

`DOUBLE COMPLEX` for `mkl_zbsrsv`.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb*lb*.

Refer to the *values* array description in BSR Format for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx

`INTEGER`. Array containing the column indices for each non-zero block in the matrix *A*.

Its length is equal to the number of non-zero blocks in the matrix *A*.

Refer to the *columns* array description in BSR Format for more details.

pntrb

`INTEGER`. Array of length *m*.

For one-based indexing: this array contains row indices, such that *pntrb*(*i*) – *pntrb*(1) + 1 is the first index of block row *i* in the array *indx*.

For zero-based indexing: this array contains row indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of block row *i* in the array *indx*.

Refer to *pointerB* array description in BSR Format for more details.

pntre

`INTEGER`. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of block row *i* in the array *indx*.

For zero-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(0) – 1 is the last index of block row *i* in the array *indx*.

Refer to *pointerE* array description in BSR Format for more details.

x

REAL for mkl_sbsrsv.

DOUBLE PRECISION for mkl_dbsrsv.

COMPLEX for mkl_cbsrsv.

DOUBLE COMPLEX for mkl_zbsrsv.

Array, size at least (*m***lb*).

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

REAL for mkl_sbsrsv.

DOUBLE PRECISION for mkl_dbsrsv.

COMPLEX for mkl_cbsrsv.

DOUBLE COMPLEX for mkl_zbsrsv.

Array, size at least (*m***lb*).

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y

Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, lb
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha
  REAL          val(*)
  REAL          x(*), y(*)
```

```
SUBROUTINE mkl_dbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, lb
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*)
  DOUBLE PRECISION        x(*), y(*)

SUBROUTINE mkl_cbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, lb
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha
  COMPLEX        val(*)
  COMPLEX        x(*), y(*)

SUBROUTINE mkl_zbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, lb
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*)
  DOUBLE COMPLEX        x(*), y(*)
```

mkl_?cscsv

Solves a system of linear equations for a sparse matrix in the CSC format (deprecated).

Syntax

call mkl_scscsv(*transa*, *m*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *x*, *y*)

call mkl_dcscsv(*transa*, *m*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *x*, *y*)

call mkl_ccscsv(*transa*, *m*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *x*, *y*)

call mkl_zcscsv(*transa*, *m*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *x*, *y*)

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?cscsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSC format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A^T)* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

> **NOTE**
> This routine supports a CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then *y* := *alpha*inv(*A*)*x*

If *transa*= 'T' or 't' or 'C' or 'c', then *y* := *alpha*inv(A^T)* *x*,

m

INTEGER. Number of columns of the matrix *A*.

alpha

REAL for mkl_scscsv.

DOUBLE PRECISION for mkl_dcscsv.

COMPLEX for mkl_ccscsv.

DOUBLE COMPLEX for mkl_zcscsv.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

REAL for mkl_scscsv.

DOUBLE PRECISION for mkl_dcscsv.

COMPLEX for mkl_ccscsv.

DOUBLE COMPLEX for mkl_zcscsv.

Array containing non-zero elements of the matrix *A*.

For one-based indexing its length is *pntre*(*m*) – *pntrb*(1).

For zero-based indexing its length is *pntre*(*m*–1) – *pntrb*(0).

Refer to *values* array description in CSC Format for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx INTEGER. Array containing the row indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *columns* array description in CSC Format for more details.

NOTE
Row indices must be sorted in increasing order for each column.

pntrb INTEGER. Array of length *m*.

For one-based indexing this array contains column indices, such that *pntrb*(*i*) – *pntrb*(1) + 1 is the first index of column *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of column *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in CSC Format for more details.

pntre INTEGER. Array of length *m*.

For one-based indexing this array contains column indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of column *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that *pntre*(*i*) – *pntrb*(1) – 1 is the last index of column *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in CSC Format for more details.

x REAL for mkl_scscsv.

DOUBLE PRECISION for mkl_dcscsv.

COMPLEX for mkl_ccscsv.

DOUBLE COMPLEX for mkl_zcscsv.

Array, size at least *m*.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

`REAL` for `mkl_scscsv`.

`DOUBLE PRECISION` for `mkl_dcscsv`.

`COMPLEX` for `mkl_ccscsv`.

`DOUBLE COMPLEX` for `mkl_zcscsv`.

Array, size at least *m*.

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y

Contains the solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha
  REAL          val(*)
  REAL          x(*), y(*)

SUBROUTINE mkl_dcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*)
  DOUBLE PRECISION        x(*), y(*)
```

```
SUBROUTINE mkl_ccscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha
  COMPLEX        val(*)
  COMPLEX        x(*), y(*)

SUBROUTINE mkl_zcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*)
  DOUBLE COMPLEX        x(*), y(*)
```

mkl_?coosv

Solves a system of linear equations for a sparse matrix in the coordinate format (deprecated).

Syntax

```
call mkl_scoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
call mkl_dcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
call mkl_ccoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
call mkl_zcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?coosv routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the coordinate format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A^T)*x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE
This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa
CHARACTER*1. Specifies the system of linear equations.
If *transa* = 'N' or 'n', then *y* := *alpha**inv(*A*)**x*
If *transa* = 'T' or 't' or 'C' or 'c', then *y* := *alpha**inv(A^T)* *x*,

m
INTEGER. Number of rows of the matrix *A*.

alpha
REAL for mkl_scoosv.
DOUBLE PRECISION for mkl_dcoosv.
COMPLEX for mkl_ccoosv.
DOUBLE COMPLEX for mkl_zcoosv.
Specifies the scalar *alpha*.

matdescra
CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val
REAL for mkl_scoosv.
DOUBLE PRECISION for mkl_dcoosv.
COMPLEX for mkl_ccoosv.
DOUBLE COMPLEX for mkl_zcoosv.
Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.
Refer to *values* array description in Coordinate Format for more details.

rowind
INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.
Refer to *rows* array description in Coordinate Format for more details.

colind
INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*.
Refer to *columns* array description in Coordinate Format for more details.

nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
x	REAL for mkl_scoosv. DOUBLE PRECISION for mkl_dcoosv. COMPLEX for mkl_ccoosv. DOUBLE COMPLEX for mkl_zcoosv. Array, size at least *m*. On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.
y	REAL for mkl_scoosv. DOUBLE PRECISION for mkl_dcoosv. COMPLEX for mkl_ccoosv. DOUBLE COMPLEX for mkl_zcoosv. Array, size at least *m*. On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y	Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  REAL          alpha
  REAL          val(*)
  REAL          x(*), y(*)
```

```
SUBROUTINE mkl_dcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*)
  DOUBLE PRECISION        x(*), y(*)

SUBROUTINE mkl_ccoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX        alpha
  COMPLEX        val(*)
  COMPLEX        x(*), y(*)

SUBROUTINE mkl_zcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*)
  DOUBLE COMPLEX        x(*), y(*)
```

mkl_?csrmm

Computes matrix - matrix product of a sparse matrix stored in the CSR format (deprecated).

Syntax

call mkl_scsrmm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_dcsrmm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_ccsrmm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_zcsrmm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

Include Files

- mkl.fi

Description

This routine is deprecated. Use Use mkl_sparse_?_mmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?csrmm routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*A^T*B + beta*C
```

or

```
C := alpha*A^H*B + beta*C,
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse row (CSR) format, A^T is the transpose of *A*, and A^H is the conjugate transpose of *A*.

> **NOTE**
> This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa
CHARACTER*1. Specifies the operation.
If *transa* = 'N' or 'n', then *C* := *alpha**A*B + *beta**C*,
If *transa* = 'T' or 't', then *C* := *alpha**A^T*B + *beta**C*,
If *transa* = 'C' or 'c', then *C* := *alpha**A^H*B + *beta**C*.

m
INTEGER. Number of rows of the matrix *A*.

n
INTEGER. Number of columns of the matrix *C*.

k
INTEGER. Number of columns of the matrix *A*.

alpha
REAL for mkl_scsrmm.
DOUBLE PRECISION for mkl_dcsrmm.
COMPLEX for mkl_ccsrmm.
DOUBLE COMPLEX for mkl_zcsrmm.
Specifies the scalar *alpha*.

matdescra — CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val — REAL for mkl_scsrmm.

DOUBLE PRECISION for mkl_dcsrmm.

COMPLEX for mkl_ccsrmm.

DOUBLE COMPLEX for mkl_zcsrmm.

Array containing non-zero elements of the matrix *A*.

For one-based indexing its length is *pntre*(m) – *pntrb*(1).

For zero-based indexing its length is *pntre*(–1) – *pntrb*(0).

Refer to *values* array description in CSR Format for more details.

indx — INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *columns* array description in CSR Format for more details.

pntrb — INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntrb*(*I*) – *pntrb*(1) + 1 is the first index of row *I* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that *pntrb*(*I*) – *pntrb*(0) is the first index of row *I* in the arrays *val* and *indx*.

Refer to *pointerb* array description in CSR Format for more details.

pntre — INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntre*(*I*) – *pntrb*(1) is the last index of row *I* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that *pntre*(*I*) – *pntrb*(0) – 1 is the last index of row *I* in the arrays *val* and *indx*.

Refer to *pointerE* array description in CSR Format for more details.

b — REAL for mkl_scsrmm.

DOUBLE PRECISION for mkl_dcsrmm.

COMPLEX for mkl_ccsrmm.

DOUBLE COMPLEX for mkl_zcsrmm.

Array, size *ldb* by at least *n* for non-transposed matrix *A* and at least *m* for transposed for one-based indexing, and (at least *k* for non-transposed matrix *A* and at least *m* for transposed, *ldb*) for zero-based indexing.

On entry with *transa*='N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb — INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

beta — REAL for mkl_scsrmm.

DOUBLE PRECISION for mkl_dcsrmm.

COMPLEX for mkl_ccsrmm.

DOUBLE COMPLEX for mkl_zcsrmm.

Specifies the scalar *beta*.

c — REAL for mkl_scsrmm.

DOUBLE PRECISION for mkl_dcsrmm.

COMPLEX for mkl_ccsrmm.

DOUBLE COMPLEX for mkl_zcsrmm.

Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.

On entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.

ldc — INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c — Overwritten by the matrix (*alpha***A***B* + *beta** *C*), (*alpha***A*^{T}**B* + *beta***C*), or (*alpha***A*^{H}**B* + *beta***C*).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha, beta
  REAL          val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha, beta
  COMPLEX        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?bsrmm

Computes matrix - matrix product of a sparse matrix stored in the BSR format (deprecated).

Syntax

call mkl_sbsrmm(*transa*, *m*, *n*, *k*, *lb*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_dbsrmm(*transa*, *m*, *n*, *k*, *lb*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_cbsrmm(*transa*, *m*, *n*, *k*, *lb*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_zbsrmm(*transa*, *m*, *n*, *k*, *lb*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

Include Files

- mkl.fi

Description

This routine is deprecated. Use Use mkl_sparse_?_mmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrmm routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*A^T*B + beta*C
```

or

```
C := alpha*A^H*B + beta*C,
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in block sparse row (BSR) format, A^T is the transpose of *A*, and A^H is the conjugate transpose of *A*.

> **NOTE**
> This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then the matrix-matrix product is computed as *C* := *alpha***A***B* + *beta***C*

If *transa* = 'T' or 't', then the matrix-vector product is computed as *C* := *alpha***A*^{T}**B* + *beta***C*

If *transa* = 'C' or 'c', then the matrix-vector product is computed as *C* := *alpha***A*^{H}**B* + *beta***C*,

m

INTEGER. Number of block rows of the matrix *A*.

n

INTEGER. Number of columns of the matrix *C*.

k

INTEGER. Number of block columns of the matrix *A*.

lb

INTEGER. Size of the block in the matrix *A*.

alpha

REAL for mkl_sbsrmm.

DOUBLE PRECISION for mkl_dbsrmm.

COMPLEX for mkl_cbsrmm.

DOUBLE COMPLEX for mkl_zbsrmm.

Specifies the scalar *alpha*.

matdescra CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val REAL for mkl_sbsrmm.

DOUBLE PRECISION for mkl_dbsrmm.

COMPLEX for mkl_cbsrmm.

DOUBLE COMPLEX for mkl_zbsrmm.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb*lb*. Refer to the *values* array description in BSR Format for more details.

indx INTEGER. Array containing the column indices for each non-zero block in the matrix *A*.

Its length is equal to the number of non-zero blocks in the matrix *A*. Refer to the *columns* array description in BSR Format for more details.

pntrb INTEGER. Array of length *m*.

For one-based indexing: this array contains row indices, such that *pntrb*(*I*) – *pntrb*(1) + 1 is the first index of block row *I* in the array *indx*.

For zero-based indexing: this array contains row indices, such that *pntrb*(*I*) – *pntrb*(0) is the first index of block row *I* in the array *indx*.

Refer to *pointerB* array description in BSR Format for more details.

pntre INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntre*(*I*) – *pntrb*(1) is the last index of block row *I* in the array *indx*.

For zero-based indexing this array contains row indices, such that *pntre*(*I*) – *pntrb*(0) – 1 is the last index of block row *I* in the array *indx*.

Refer to *pointerE* array description in BSR Format for more details.

b REAL for mkl_sbsrmm.

DOUBLE PRECISION for mkl_dbsrmm.

COMPLEX for mkl_cbsrmm.

DOUBLE COMPLEX for mkl_zbsrmm.

Array, size *ldb* by at least *n* for non-transposed matrix *A* and at least *m* for transposed for one-based indexing, and (at least *k* for non-transposed matrix *A* and at least *m* for transposed, *ldb*) for zero-based indexing.

	On entry with *transa*='N' or 'n', the leading *n*-by-*k* block part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* block part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the leading dimension (in blocks) of *b* as declared in the calling (sub)program.
beta	REAL for mkl_sbsrmm. DOUBLE PRECISION for mkl_dbsrmm. COMPLEX for mkl_cbsrmm. DOUBLE COMPLEX for mkl_zbsrmm. Specifies the scalar *beta*.
c	REAL for mkl_sbsrmm. DOUBLE PRECISION for mkl_dbsrmm. COMPLEX for mkl_cbsrmm. DOUBLE COMPLEX for mkl_zbsrmm. Array, size (*ldc*, *n*) for one-based indexing, size (*k*, *ldc*) for zero-based indexing. On entry, the leading *m*-by-*n* block part of the array *c* must contain the matrix *C*, otherwise the leading *n*-by-*k* block part of the array *c* must contain the matrix *C*.
ldc	INTEGER. Specifies the leading dimension (in blocks) of *c* as declared in the calling (sub)program.

Output Parameters

c	Overwritten by the matrix (*alpha***A***B* + *beta***C*) or (*alpha***A*^{T}**B* + *beta***C*) or (*alpha***A*^{H}**B* + *beta***C*).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrmm(transa, m, n, k, lb, alpha, matdescra, val,
 indx, pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ld, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha, beta
  REAL          val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dbsrmm(transa, m, n, k, lb, alpha, matdescra, val,
 indx, pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ld, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cbsrmm(transa, m, n, k, lb, alpha, matdescra, val,
 indx, pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ld, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha, beta
  COMPLEX        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zbsrmm(transa, m, n, k, lb, alpha, matdescra, val,
 indx, pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ld, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?cscmm

Computes matrix-matrix product of a sparse matrix stored in the CSC format (deprecated).

Syntax

call mkl_scscmm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_dcscmm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_ccscmm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_zcscmm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *indx*, *pntrb*, *pntre*, *b*, *ldb*, *beta*, *c*, *ldc*)

Include Files

- mkl.fi

Description

This routine is deprecated. Use Use mkl_sparse_?_mmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?cscmm routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*A^T*B + beta*C,
```

or

```
C := alpha*A^H*B + beta*C,
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse column (CSC) format, A^T is the transpose of *A*, and A^H is the conjugate transpose of *A*.

NOTE
This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then *C* := *alpha*A* B + beta*C*

If *transa* = 'T' or 't', then *C* := *alpha*A^T*B + beta*C*,

If *transa* ='C' or 'c', then *C* := *alpha*A^H*B + beta*C*

m

INTEGER. Number of rows of the matrix *A*.

n

INTEGER. Number of columns of the matrix *C*.

k

INTEGER. Number of columns of the matrix *A*.

alpha

REAL for mkl_scscmm.

DOUBLE PRECISION for mkl_dcscmm.

COMPLEX for mkl_ccscmm.

DOUBLE COMPLEX for mkl_zcscmm.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

REAL for mkl_scscmm.

DOUBLE PRECISION for mkl_dcscmm.

COMPLEX for mkl_ccscmm.

DOUBLE COMPLEX for mkl_zcscmm.

Array containing non-zero elements of the matrix *A*.

For one-based indexing its length is *pntrb*(*k*) – *pntrb*(1).

For zero-based indexing its length is *pntre*(*m*–1) – *pntrb*(0).

Refer to *values* array description in CSC Format for more details.

indx

INTEGER. Array containing the row indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *rows* array description in CSC Format for more details.

pntrb

INTEGER. Array of length *k*.

For one-based indexing this array contains column indices, such that *pntrb*(*i*) – *pntrb*(1) + 1 is the first index of column *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of column *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in CSC Format for more details.

pntre

INTEGER. Array of length *k*.

For one-based indexing this array contains column indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of column *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that *pntre*(*i*) – *pntrb*(1) – 1 is the last index of column *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in CSC Format for more details.

b

REAL for mkl_scscmm.

DOUBLE PRECISION for mkl_dcscmm.

COMPLEX for mkl_ccscmm.

DOUBLE COMPLEX for mkl_zcscmm.

Array, size *ldb* by at least *n* for non-transposed matrix *A* and at least *m* for transposed for one-based indexing, and (at least *k* for non-transposed matrix *A* and at least *m* for transposed, *ldb*) for zero-based indexing.

On entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb — INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

beta — REAL*8. Specifies the scalar *beta*.

c — REAL for mkl_scscmm.

DOUBLE PRECISION for mkl_dcscmm.

COMPLEX for mkl_ccscmm.

DOUBLE COMPLEX for mkl_zcscmm.

Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.

On entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.

ldc — INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c — Overwritten by the matrix (*alpha***A***B* + *beta** *C*) or (*alpha***A*^{T}**B* + *beta***C*) or (*alpha***A*^{H}**B* + *beta***C*).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       indx(*), pntrb(k), pntre(k)
  REAL          alpha, beta
  REAL          val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcscmm(transa, m, n, k, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       indx(*), pntrb(k), pntre(k)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_ccscmm(transa, m, n, k, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       indx(*), pntrb(k), pntre(k)
  COMPLEX        alpha, beta
  COMPLEX        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zcscmm(transa, m, n, k, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       indx(*), pntrb(k), pntre(k)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?coomm

Computes matrix-matrix product of a sparse matrix stored in the coordinate format (deprecated).

Syntax

call mkl_scoomm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *rowind*, *colind*, *nnz*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_dcoomm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *rowind*, *colind*, *nnz*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_ccoomm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *rowind*, *colind*, *nnz*, *b*, *ldb*, *beta*, *c*, *ldc*)

call mkl_zcoomm(*transa*, *m*, *n*, *k*, *alpha*, *matdescra*, *val*, *rowind*, *colind*, *nnz*, *b*, *ldb*, *beta*, *c*, *ldc*)

Include Files

- mkl.fi

Description

This routine is deprecated. Use Use mkl_sparse_?_mmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?coomm routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*A^T*B + beta*C,
```

or

```
C := alpha*A^H*B + beta*C,
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the coordinate format, A^T is the transpose of *A*, and A^H is the conjugate transpose of *A*.

> **NOTE**
> This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then *C* := *alpha***A***B* + *beta***C*

If *transa* = 'T' or 't', then *C* := *alpha***A*^{T}**B* + *beta***C*,

If *transa* = 'C' or 'c', then *C* := *alpha***A*^{H}**B* + *beta***C*.

m

INTEGER. Number of rows of the matrix *A*.

n

INTEGER. Number of columns of the matrix *C*.

k

INTEGER. Number of columns of the matrix *A*.

alpha

REAL for mkl_scoomm.

DOUBLE PRECISION for mkl_dcoomm.

COMPLEX for mkl_ccoomm.

DOUBLE COMPLEX for mkl_zcoomm.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

REAL for mkl_scoomm.

DOUBLE PRECISION for mkl_dcoomm.

COMPLEX for mkl_ccoomm.

DOUBLE COMPLEX for mkl_zcoomm.

Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.

Refer to *values* array description in Coordinate Format for more details.

rowind

INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in Coordinate Format for more details.

colind

INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*.

Refer to *columns* array description in Coordinate Format for more details.

nnz

INTEGER. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in Coordinate Format for more details.

b

REAL for mkl_scoomm.

DOUBLE PRECISION for mkl_dcoomm.

COMPLEX for mkl_ccoomm.

DOUBLE COMPLEX for mkl_zcoomm.

Array, size *ldb* by at least *n* for non-transposed matrix *A* and at least *m* for transposed for one-based indexing, and (at least *k* for non-transposed matrix *A* and at least *m* for transposed, *ldb*) for zero-based indexing.

On entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

beta

REAL for mkl_scoomm.

DOUBLE PRECISION for mkl_dcoomm.

COMPLEX for mkl_ccoomm.

DOUBLE COMPLEX for mkl_zcoomm.

Specifies the scalar *beta*.

c	REAL for mkl_scoomm. DOUBLE PRECISION for mkl_dcoomm. COMPLEX for mkl_ccoomm. DOUBLE COMPLEX for mkl_zcoomm. Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing. On entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.
ldc	INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c	Overwritten by the matrix (*alpha*A*B + beta*C*), (*alpha*A*^{T}**B + beta*C*), or (*alpha*A*^{H}**B + beta*C*).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoomm(transa, m, n, k, alpha, matdescra, val,
 rowind, colind, nnz, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  REAL          alpha, beta
  REAL          val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dcoomm(transa, m, n, k, alpha, matdescra, val,
 rowind, colind, nnz, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ccoomm(transa, m, n, k, alpha, matdescra, val,
 rowind, colind, nnz, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX        alpha, beta
  COMPLEX        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zcoomm(transa, m, n, k, alpha, matdescra, val,
 rowind, colind, nnz, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?csrsm

Solves a system of linear matrix equations for a sparse matrix in the CSR format (deprecated).

Syntax

```
call mkl_scsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
call mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
call mkl_ccsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
call mkl_zcsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?csrsm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSR format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A^T)*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE
This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

`CHARACTER*1`. Specifies the system of linear equations.

If *transa* = `'N'` or `'n'`, then *C* := *alpha**inv(*A*)**B*

If *transa* = `'T'` or `'t'` or `'C'` or `'c'`, then *C* := *alpha**inv(A^T)**B*,

m

`INTEGER`. Number of columns of the matrix *A*.

n

`INTEGER`. Number of columns of the matrix *C*.

alpha

`REAL` for `mkl_scsrsm`.

`DOUBLE PRECISION` for `mkl_dcsrsm`.

`COMPLEX` for `mkl_ccsrsm`.

`DOUBLE COMPLEX` for `mkl_zcsrsm`.

Specifies the scalar *alpha*.

matdescra

`CHARACTER`. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

`REAL` for `mkl_scsrsm`.

`DOUBLE PRECISION` for `mkl_dcsrsm`.

`COMPLEX` for `mkl_ccsrsm`.

`DOUBLE COMPLEX` for `mkl_zcsrsm`.

Array containing non-zero elements of the matrix *A*.

For one-based indexing its length is *pntre*(m) – *pntrb*(1).

For zero-based indexing its length is *pntre*(m–1) – *pntrb*(0).

Refer to *values* array description in CSR Format for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *columns* array description in CSR Format for more details.

NOTE
Column indices must be sorted in increasing order for each row.

pntrb INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntrb*(*i*) – *pntrb*(1) + 1 is the first index of row *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of row *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in CSR Format for more details.

pntre INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of row *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(0) – 1 is the last index of row *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in CSR Format for more details.

b REAL for mkl_scsrsm.

DOUBLE PRECISION for mkl_dcsrsm.

COMPLEX for mkl_ccsrsm.

DOUBLE COMPLEX for mkl_zcsrsm.

Array, size (*ldb*, *n*) for one-based indexing, and (*m*, *ldb*) for zero-based indexing.

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

ldc INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c REAL*8.

Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.

The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsm(transa, m, n, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha
  REAL          val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_ccsrsm(transa, m, n, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha
  COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zcsrsm(transa, m, n, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?cscsm

Solves a system of linear matrix equations for a sparse matrix in the CSC format (deprecated).

Syntax

```
call mkl_scscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
call mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
call mkl_ccscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
call mkl_zcscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?cscsm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSC format:

$C := alpha*inv(A)*B$

or

$C := alpha*inv(A^T)*B$,

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

> **NOTE**
> This routine supports a CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa
CHARACTER*1. Specifies the system of equations.
If *transa* = 'N' or 'n', then *C* := *alpha**inv(*A*)**B*
If *transa* = 'T' or 't' or 'C' or 'c', then *C* := *alpha**inv(A^T)**B*,

m
INTEGER. Number of columns of the matrix *A*.

n
INTEGER. Number of columns of the matrix *C*.

alpha
REAL for mkl_scscsm.
DOUBLE PRECISION for mkl_dcscsm.
COMPLEX for mkl_ccscsm.
DOUBLE COMPLEX for mkl_zcscsm.
Specifies the scalar *alpha*.

matdescra
CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val
REAL for mkl_scscsm.
DOUBLE PRECISION for mkl_dcscsm.
COMPLEX for mkl_ccscsm.
DOUBLE COMPLEX for mkl_zcscsm.
Array containing non-zero elements of the matrix *A*.
For one-based indexing its length is *pntre*(*k*) - *pntrb*(1).
For zero-based indexing its length is *pntre*(*m*-1) - *pntrb*(0).
Refer to *values* array description in CSC Format for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).
No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx
INTEGER. Array containing the row indices for each non-zero element of the matrix *A*. Its length is equal to length of the *val* array.
Refer to *rows* array description in CSC Format for more details.

NOTE
Row indices must be sorted in increasing order for each column.

pntrb
INTEGER. Array of length *m*.

For one-based indexing this array contains column indices, such that *pntrb*(*I*) – *pntrb*(1) + 1 is the first index of column *I* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that *pntrb*(*I*) – *pntrb*(0) is the first index of column *I* in the arrays *val* and *indx*.

Refer to *pointerb* array description in CSC Format for more details.

pntre

INTEGER. Array of length *m*.

For one-based indexing this array contains column indices, such that *pntre*(*I*) – *pntrb*(1) is the last index of column *I* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that *pntre*(*I*) – *pntrb*(1)–1 is the last index of column *I* in the arrays *val* and *indx*.

Refer to *pointerE* array description in CSC Format for more details.

b

REAL for mkl_scscsm.

DOUBLE PRECISION for mkl_dcscsm.

COMPLEX for mkl_ccscsm.

DOUBLE COMPLEX for mkl_zcscsm.

Array, size *ldb* by *n* for one-based indexing, and (*m*, *ldb*) for zero-based indexing.

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

ldc

INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c

REAL for mkl_scscsm.

DOUBLE PRECISION for mkl_dcscsm.

COMPLEX for mkl_ccscsm.

DOUBLE COMPLEX for mkl_zcscsm.

Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.

The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscsm(transa, m, n, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha
  REAL          val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_ccscsm(transa, m, n, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha
  COMPLEX        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zcscsm(transa, m, n, alpha, matdescra, val, indx,
 pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?coosm

Solves a system of linear matrix equations for a sparse matrix in the coordinate format (deprecated).

Syntax

```
call mkl_scoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
call mkl_dcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
call mkl_ccoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
call mkl_zcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?coosm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the coordinate format:

$C := alpha*inv(A)*B$

or

$C := alpha*inv(A^T)*B$,

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE
This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the system of linear equations.

If *transa* = 'N' or 'n', then the matrix-matrix product is computed as $C := alpha*inv(A)*B$

If *transa* = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $C := alpha*inv(A^T)*B$,

m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
alpha	REAL for mkl_scoosm. DOUBLE PRECISION for mkl_dcoosm. COMPLEX for mkl_ccoosm. DOUBLE COMPLEX for mkl_zcoosm. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".
val	REAL for mkl_scoosm. DOUBLE PRECISION for mkl_dcoosm. COMPLEX for mkl_ccoosm. DOUBLE COMPLEX for mkl_zcoosm. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order. Refer to *values* array description in Coordinate Format for more details.
rowind	INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*. Refer to *rows* array description in Coordinate Format for more details.
colind	INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.
nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
b	REAL for mkl_scoosm. DOUBLE PRECISION for mkl_dcoosm. COMPLEX for mkl_ccoosm. DOUBLE COMPLEX for mkl_zcoosm. Array, size *ldb* by *n* for one-based indexing, and (*m*, *ldb*) for zero-based indexing. Before entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

ldc INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c REAL for mkl_scoosm.

DOUBLE PRECISION for mkl_dcoosm.

COMPLEX for mkl_ccoosm.

DOUBLE COMPLEX for mkl_zcoosm.

Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.

The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  REAL         alpha
  REAL          val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ccoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX        alpha
  COMPLEX       val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?bsrsm

Solves a system of linear matrix equations for a sparse matrix in the BSR format (deprecated).

Syntax

`call mkl_scsrsm(`*transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc*`)`

`call mkl_dcsrsm(`*transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc*`)`

`call mkl_ccsrsm(`*transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc*`)`

`call mkl_zcsrsm(`*transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc*`)`

Include Files

- `mkl.fi`

Description

This routine is deprecated. Use mkl_sparse_?_trsmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?bsrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the BSR format:

$C := alpha*\mathrm{inv}(A)*B$

or

$C := alpha*\mathrm{inv}(A^T)*B,$

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE
This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

`CHARACTER*1`. Specifies the operation.

If *transa* = `'N'` or `'n'`, then the matrix-matrix product is computed as *C* := *alpha*`*inv(`*A*`)*`*B*.

If *transa* = `'T'` or `'t'` or `'C'` or `'c'`, then the matrix-vector product is computed as *C* := *alpha*`*inv(`A^T`)*`*B*.

m

`INTEGER`. Number of block columns of the matrix *A*.

n

`INTEGER`. Number of columns of the matrix *C*.

lb

`INTEGER`. Size of the block in the matrix *A*.

alpha

`REAL` for `mkl_sbsrsm`.

`DOUBLE PRECISION` for `mkl_dbsrsm`.

`COMPLEX` for `mkl_cbsrsm`.

`DOUBLE COMPLEX` for `mkl_zbsrsm`.

Specifies the scalar *alpha*.

matdescra

`CHARACTER`. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

`REAL` for `mkl_sbsrsm`.

`DOUBLE PRECISION` for `mkl_dbsrsm`.

`COMPLEX` for `mkl_cbsrsm`.

`DOUBLE COMPLEX` for `mkl_zbsrsm`.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb*`*`*lb*. Refer to the *values* array description in BSR Format for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the number of non-zero blocks in the matrix *A*.

Refer to the *columns* array description in BSR Format for more details.

pntrb INTEGER. Array of length *m*.

For one-based indexing: this array contains row indices, such that *pntrb*(*i*) – *pntrb*(1) + 1 is the first index of block row *i* in the array *indx*.

For zero-based indexing: this array contains row indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of block row *i* in the array *indx*.

Refer to *pointerB* array description in BSR Format for more details.

pntre INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of block row *i* in the array *indx*.

For zero-based indexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(0) – 1 is the last index of block row *i* in the array *indx*.

Refer to *pointerE* array description in BSR Format for more details.

b REAL for mkl_sbsrsm.

DOUBLE PRECISION for mkl_dbsrsm.

COMPLEX for mkl_cbsrsm.

DOUBLE COMPLEX for mkl_zbsrsm.

Array, size (*ldb*, *n*) for one-based indexing, size (*m*, *ldb*) for zero-based indexing.

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the leading dimension (in blocks) of *b* as declared in the calling (sub)program.

ldc INTEGER. Specifies the leading dimension (in blocks) of *c* as declared in the calling (sub)program.

Output Parameters

c REAL for mkl_sbsrsm.

DOUBLE PRECISION for mkl_dbsrsm.

`COMPLEX` for `mkl_cbsrsm`.

`DOUBLE COMPLEX` for `mkl_zbsrsm`.

Array, size `(ldc, n)` for one-based indexing, size `(m, ldc)` for zero-based indexing.

The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
  CHARACTER*1     transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, lb, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL          alpha
  REAL          val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
  CHARACTER*1     transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, lb, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
  CHARACTER*1     transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, lb, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  COMPLEX        alpha
  COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
  CHARACTER*1     transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, lb, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?diamv

Computes matrix - vector product for a sparse matrix in the diagonal format with one-based indexing (deprecated).

Syntax

```
call mkl_sdiamv(transa, m, k, alpha, matdescra, val, lval, idiag, ndiag, x, beta, y)
call mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, idiag, ndiag, x, beta, y)
call mkl_cdiamv(transa, m, k, alpha, matdescra, val, lval, idiag, ndiag, x, beta, y)
call mkl_zdiamv(transa, m, k, alpha, matdescra, val, lval, idiag, ndiag, x, beta, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?diamv routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A^T*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored in the diagonal format, A^T is the transpose of *A*.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then $y := alpha*A*x + beta*y$,

If *transa* = 'T' or 't' or 'C' or 'c', then $y := alpha*A^{T}*x + beta*y$.

m

INTEGER. Number of rows of the matrix *A*.

k

INTEGER. Number of columns of the matrix *A*.

alpha

REAL for mkl_sdiamv.

DOUBLE PRECISION for mkl_ddiamv.

COMPLEX for mkl_cdiamv.

DOUBLE COMPLEX for mkl_zdiamv.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val

REAL for mkl_sdiamv.

DOUBLE PRECISION for mkl_ddiamv.

COMPLEX for mkl_cdiamv.

DOUBLE COMPLEX for mkl_zdiamv.

Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.

lval

INTEGER. Leading dimension of *val*, $lval \geq m$. Refer to *lval* description in Diagonal Storage Scheme for more details.

idiag

INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*.

Refer to *distance* array description in Diagonal Storage Scheme for more details.

ndiag

INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.

x

REAL for mkl_sdiamv.

DOUBLE PRECISION for mkl_ddiamv.

COMPLEX for mkl_cdiamv.

DOUBLE COMPLEX for mkl_zdiamv.

	Array, size at least *k* if *transa* = 'N' or 'n', and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.
beta	REAL for mkl_sdiamv. DOUBLE PRECISION for mkl_ddiamv. COMPLEX for mkl_cdiamv. DOUBLE COMPLEX for mkl_zdiamv. Specifies the scalar *beta*.
y	REAL for mkl_sdiamv. DOUBLE PRECISION for mkl_ddiamv. COMPLEX for mkl_cdiamv. DOUBLE COMPLEX for mkl_zdiamv. Array, size at least *m* if *transa* = 'N' or 'n', and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y	Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, lval, ndiag
  INTEGER      idiag(*)
  REAL          alpha, beta
  REAL          val(lval,*), x(*), y(*)

SUBROUTINE mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE PRECISION          alpha, beta
  DOUBLE PRECISION          val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, lval, ndiag
  INTEGER      idiag(*)
  COMPLEX          alpha, beta
  COMPLEX          val(lval,*), x(*), y(*)

SUBROUTINE mkl_zdiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE COMPLEX          alpha, beta
  DOUBLE COMPLEX          val(lval,*), x(*), y(*)
```

mkl_?skymv

Computes matrix - vector product for a sparse matrix in the skyline storage format with one-based indexing (deprecated).

Syntax

```
call mkl_sskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
call mkl_dskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
call mkl_cskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
call mkl_zskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?skymv routine performs a matrix-vector operation defined as

$y := alpha*A*x + beta*y$

or

$y := alpha*A^{T}*x + beta*y,$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored using the skyline storage scheme, A^T is the transpose of *A*.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

`CHARACTER*1`. Specifies the operation.

If *transa* = `'N'` or `'n'`, then *y* := *alpha***A***x* + *beta***y*

If *transa* = `'T'` or `'t'` or `'C'` or `'c'`, then *y* := *alpha***A*^{T}**x* + *beta***y*,

m

`INTEGER`. Number of rows of the matrix *A*.

k

`INTEGER`. Number of columns of the matrix *A*.

alpha

`REAL` for `mkl_sskymv`.

`DOUBLE PRECISION` for `mkl_dskymv`.

`COMPLEX` for `mkl_cskymv`.

`DOUBLE COMPLEX` for `mkl_zskymv`.

Specifies the scalar *alpha*.

matdescra

`CHARACTER`. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

NOTE
General matrices (*matdescra*`(1)='G'`) is not supported.

val

`REAL` for `mkl_sskymv`.

`DOUBLE PRECISION` for `mkl_dskymv`.

`COMPLEX` for `mkl_cskymv`.

`DOUBLE COMPLEX` for `mkl_zskymv`.

Array containing the set of elements of the matrix *A* in the skyline profile form.

If *matdescrsa*`(2)= 'L'`, then *val* contains elements from the low triangle of the matrix *A*.

If *matdescrsa*(2)= 'U', then *val* contains elements from the upper triangle of the matrix *A*.

Refer to *values* array description in Skyline Storage Scheme for more details.

pntr

INTEGER. Array of length (*m* + 1) for lower triangle, and (*k* + 1) for upper triangle.

It contains the indices specifying in the *val* the positions of the first element in each row (column) of the matrix *A*. Refer to *pointers* array description in Skyline Storage Scheme for more details.

x

REAL for mkl_sskymv.

DOUBLE PRECISION for mkl_dskymv.

COMPLEX for mkl_cskymv.

DOUBLE COMPLEX for mkl_zskymv.

Array, size at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta

REAL for mkl_sskymv.

DOUBLE PRECISION for mkl_dskymv.

COMPLEX for mkl_cskymv.

DOUBLE COMPLEX for mkl_zskymv.

Specifies the scalar *beta*.

y

REAL for mkl_sskymv.

DOUBLE PRECISION for mkl_dskymv.

COMPLEX for mkl_cskymv.

DOUBLE COMPLEX for mkl_zskymv.

Array, size at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y

Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k
  INTEGER       pntr(*)
  REAL          alpha, beta
  REAL          val(*), x(*), y(*)

SUBROUTINE mkl_dskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k
  INTEGER       pntr(*)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), x(*), y(*)

SUBROUTINE mkl_cdskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k
  INTEGER       pntr(*)
  COMPLEX        alpha, beta
  COMPLEX        val(*), x(*), y(*)

SUBROUTINE mkl_zskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, k
  INTEGER       pntr(*)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), x(*), y(*)
```

mkl_?diasv

Solves a system of linear equations for a sparse matrix in the diagonal format with one-based indexing (deprecated).

Syntax

```
call mkl_sdiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
call mkl_ddiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
call mkl_cdiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
call mkl_zdiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?diasv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

$y := alpha*inv(A)*x$

or

$y := alpha*inv(A^T)* x,$

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the system of linear equations.

If *transa* = 'N' or 'n', then $y := alpha*inv(A)*x$

If *transa* = 'T' or 't' or 'C' or 'c', then $y := alpha*inv(A^T)*x$,

m

INTEGER. Number of rows of the matrix *A*.

alpha

REAL for mkl_sdiasv.

DOUBLE PRECISION for mkl_ddiasv.

COMPLEX for mkl_cdiasv.

DOUBLE COMPLEX for mkl_zdiasv.

Specifies the scalar *alpha*.

matdescra
CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val
REAL for mkl_sdiasv.

DOUBLE PRECISION for mkl_ddiasv.

COMPLEX for mkl_cdiasv.

DOUBLE COMPLEX for mkl_zdiasv.

Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.

lval
INTEGER. Leading dimension of *val*, *lval*≥*m*. Refer to *lval* description in Diagonal Storage Scheme for more details.

idiag
INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*.

NOTE
All elements of this array must be sorted in increasing order.

Refer to *distance* array description in Diagonal Storage Scheme for more details.

ndiag
INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.

x
REAL for mkl_sdiasv.

DOUBLE PRECISION for mkl_ddiasv.

COMPLEX for mkl_cdiasv.

DOUBLE COMPLEX for mkl_zdiasv.

Array, size at least *m*.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y
REAL for mkl_sdiasv.

DOUBLE PRECISION for mkl_ddiasv.

COMPLEX for mkl_cdiasv.

DOUBLE COMPLEX for mkl_zdiasv.

Array, size at least *m*.

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y	Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, lval, ndiag
  INTEGER       indiag(*)
  REAL        alpha
  REAL        val(lval,*), x(*), y(*)

SUBROUTINE mkl_ddiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, lval, ndiag
  INTEGER       indiag(*)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(lval,*), x(*), y(*)

SUBROUTINE mkl_cdiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, lval, ndiag
  INTEGER       indiag(*)
  COMPLEX        alpha
  COMPLEX        val(lval,*), x(*), y(*)

SUBROUTINE mkl_zdiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, lval, ndiag
  INTEGER       indiag(*)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(lval,*), x(*), y(*)
```

mkl_?skysv

Solves a system of linear equations for a sparse matrix in the skyline format with one-based indexing (deprecated).

Syntax

```
call mkl_sskysv(transa, m, alpha, matdescra, val, pntr, x, y)
call mkl_dskysv(transa, m, alpha, matdescra, val, pntr, x, y)
call mkl_cskysv(transa, m, alpha, matdescra, val, pntr, x, y)
call mkl_zskysv(transa, m, alpha, matdescra, val, pntr, x, y)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsvfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?skysv routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the skyline storage format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A^T)*x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa

CHARACTER*1. Specifies the system of linear equations.

If *transa* = 'N' or 'n', then *y* := *alpha*inv(*A*)*x*

If *transa* = 'T' or 't' or 'C' or 'c', then *y* := *alpha*inv(A^T)* *x*,

m

INTEGER. Number of rows of the matrix *A*.

alpha

REAL for mkl_sskysv.

DOUBLE PRECISION for mkl_dskysv.

COMPLEX for mkl_cskysv.

DOUBLE COMPLEX for mkl_zskysv.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

NOTE
General matrices (*matdescra*(1)='G') is not supported.

val

REAL for mkl_sskysv.

DOUBLE PRECISION for mkl_dskysv.

COMPLEX for mkl_cskysv.

DOUBLE COMPLEX for mkl_zskysv.

Array containing the set of elements of the matrix *A* in the skyline profile form.

If *matdescra*(2)= 'L', then *val* contains elements from the low triangle of the matrix *A*.

If *matdescra*(2)= 'U', then *val* contains elements from the upper triangle of the matrix *A*.

Refer to *values* array description in Skyline Storage Scheme for more details.

pntr

INTEGER. Array of length (*m* + 1) for lower triangle, and (*k* + 1) for upper triangle.

It contains the indices specifying in the *val* the positions of the first element in each row (column) of the matrix *A*. Refer to *pointers* array description in Skyline Storage Scheme for more details.

x

REAL for mkl_sskysv.

DOUBLE PRECISION for mkl_dskysv.

COMPLEX for mkl_cskysv.

DOUBLE COMPLEX for mkl_zskysv.

Array, size at least *m*.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

REAL for mkl_sskysv.

DOUBLE PRECISION for mkl_dskysv.

COMPLEX for mkl_cskysv.

DOUBLE COMPLEX for mkl_zskysv.

Array, size at least *m*.

	On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y	Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskysv(transa, m, alpha, matdescra, val, pntr, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       pntr(*)
  REAL        alpha
  REAL        val(*), x(*), y(*)

SUBROUTINE mkl_dskysv(transa, m, alpha, matdescra, val, pntr, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       pntr(*)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*), x(*), y(*)

SUBROUTINE mkl_cskysv(transa, m, alpha, matdescra, val, pntr, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       pntr(*)
  COMPLEX        alpha
  COMPLEX        val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zskysv(transa, m, alpha, matdescra, val, pntr, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m
  INTEGER       pntr(*)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*), x(*), y(*)
```

mkl_?diamm

Computes matrix-matrix product of a sparse matrix stored in the diagonal format with one-based indexing (deprecated).

Syntax

call mkl_sdiamm(*transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, beta, c, ldc*)

call mkl_ddiamm(*transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, beta, c, ldc*)

call mkl_cdiamm(*transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, beta, c, ldc*)

call mkl_zdiamm(*transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, beta, c, ldc*)

Include Files

- mkl.fi

Description

This routine is deprecated. Use Use mkl_sparse_?_mmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?diamm routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*A^T*B + beta*C,
```

or

```
C := alpha*A^H*B + beta*C,
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the diagonal format, A^T is the transpose of *A*, and A^H is the conjugate transpose of *A*.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa
CHARACTER*1. Specifies the operation.
If *transa* = 'N' or 'n', then $C := alpha*A*B + beta*C$,
If *transa* = 'T' or 't', then $C := alpha*A^{T}*B + beta*C$,
If *transa* = 'C' or 'c', then $C := alpha*A^{H}*B + beta*C$.

m
INTEGER. Number of rows of the matrix *A*.

n
INTEGER. Number of columns of the matrix *C*.

k
INTEGER. Number of columns of the matrix *A*.

alpha
REAL for mkl_sdiamm.
DOUBLE PRECISION for mkl_ddiamm.
COMPLEX for mkl_cdiamm.
DOUBLE COMPLEX for mkl_zdiamm.
Specifies the scalar *alpha*.

matdescra
CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

val
REAL for mkl_sdiamm.
DOUBLE PRECISION for mkl_ddiamm.
COMPLEX for mkl_cdiamm.
DOUBLE COMPLEX for mkl_zdiamm.
Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.

lval
INTEGER. Leading dimension of *val*, *lval*≥*m*. Refer to *lval* description in Diagonal Storage Scheme for more details.

idiag
INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*.
Refer to *distance* array description in Diagonal Storage Scheme for more details.

ndiag
INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.

b

REAL for mkl_sdiamm.

DOUBLE PRECISION for mkl_ddiamm.

COMPLEX for mkl_cdiamm.

DOUBLE COMPLEX for mkl_zdiamm.

Array, size (*ldb*, *n*).

On entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

beta

REAL for mkl_sdiamm.

DOUBLE PRECISION for mkl_ddiamm.

COMPLEX for mkl_cdiamm.

DOUBLE COMPLEX for mkl_zdiamm.

Specifies the scalar *beta*.

c

REAL for mkl_sdiamm.

DOUBLE PRECISION for mkl_ddiamm.

COMPLEX for mkl_cdiamm.

DOUBLE COMPLEX for mkl_zdiamm.

Array, size *ldc* by *n*.

On entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.

ldc

INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c

Overwritten by the matrix (*alpha***A***B* + *beta***C*), (*alpha***A*^{T}**B* + *beta***C*), or (*alpha***A*^{H}**B* + *beta***C*).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiamm(transa, m, n, k, alpha, matdescra, val, lval,
 idiag, ndiag, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc, lval, ndiag
  INTEGER      idiag(*)
  REAL          alpha, beta
  REAL          val(lval,*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_ddiamm(transa, m, n, k, alpha, matdescra, val, lval,
 idiag, ndiag, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(lval,*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cdiamm(transa, m, n, k, alpha, matdescra, val, lval,
 idiag, ndiag, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc, lval, ndiag
  INTEGER      idiag(*)
  COMPLEX        alpha, beta
  COMPLEX        val(lval,*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zdiamm(transa, m, n, k, alpha, matdescra, val, lval,
 idiag, ndiag, b, ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(lval,*), b(ldb,*), c(ldc,*)
```

mkl_?skymm

Computes matrix-matrix product of a sparse matrix stored using the skyline storage scheme with one-based indexing (deprecated).

Syntax

```
call mkl_sskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta, c, ldc)
call mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta, c, ldc)
call mkl_cskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta, c, ldc)
call mkl_zskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta, c, ldc)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use Use mkl_sparse_?_mmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?skymm routine performs a matrix-matrix operation defined as

$C := alpha*A*B + beta*C$

or

$C := alpha*A^{T}*B + beta*C,$

or

$C := alpha*A^{H}*B + beta*C,$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the skyline storage format, A^{T} is the transpose of *A*, and A^{H} is the conjugate transpose of *A*.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa — CHARACTER*1. Specifies the operation.

If *transa* = 'N' or 'n', then $C := alpha*A*B + beta*C$,

If *transa* = 'T' or 't', then $C := alpha*A^{T}*B + beta*C$,

If *transa* = 'C' or 'c', then $C := alpha*A^{H}*B + beta*C$.

m — INTEGER. Number of rows of the matrix *A*.

n — INTEGER. Number of columns of the matrix *C*.

k — INTEGER. Number of columns of the matrix *A*.

alpha — REAL for mkl_sskymm.

DOUBLE PRECISION for mkl_dskymm.

COMPLEX for mkl_cskymm.

DOUBLE COMPLEX for mkl_zskymm.

Specifies the scalar *alpha*.

matdescra — CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

NOTE
General matrices (*matdescra* (1)='G') is not supported.

val — REAL for mkl_sskymm.

DOUBLE PRECISION for mkl_dskymm.

COMPLEX for mkl_cskymm.

DOUBLE COMPLEX for mkl_zskymm.

Array containing the set of elements of the matrix *A* in the skyline profile form.

If *matdescrsa*(2)= 'L', then *val* contains elements from the low triangle of the matrix *A*.

If *matdescrsa*(2)= 'U', then *val* contains elements from the upper triangle of the matrix *A*.

Refer to *values* array description in Skyline Storage Scheme for more details.

pntr — INTEGER. Array of length (*m* + 1) for lower triangle, and (*k* + 1) for upper triangle.

It contains the indices specifying the positions of the first element of the matrix *A* in each row (for the lower triangle) or column (for upper triangle) in the *val* array. Refer to *pointers* array description in Skyline Storage Scheme for more details.

b — REAL for mkl_sskymm.

DOUBLE PRECISION for mkl_dskymm.

COMPLEX for mkl_cskymm.

DOUBLE COMPLEX for mkl_zskymm.

Array, size (*ldb*, *n*).

On entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb — INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

beta — REAL for mkl_sskymm.

DOUBLE PRECISION for mkl_dskymm.

COMPLEX for mkl_cskymm.

DOUBLE COMPLEX for mkl_zskymm.

Specifies the scalar *beta*.

c — REAL for mkl_sskymm.

DOUBLE PRECISION for mkl_dskymm.

COMPLEX for mkl_cskymm.

DOUBLE COMPLEX for mkl_zskymm.

Array, size *ldc* by *n*.

On entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.

ldc — INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c — Overwritten by the matrix (*alpha***A***B* + *beta***C*), (*alpha***A*^{T}**B* + *beta***C*), or (*alpha***A*^{H}**B* + *beta***C*).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
 ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       pntr(*)
  REAL          alpha, beta
  REAL          val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
 ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       pntr(*)
  DOUBLE PRECISION        alpha, beta
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
 ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       pntr(*)
  COMPLEX        alpha, beta
  COMPLEX        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
 ldb, beta, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, k, ldb, ldc
  INTEGER       pntr(*)
  DOUBLE COMPLEX        alpha, beta
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?diasm

Solves a system of linear matrix equations for a sparse matrix in the diagonal format with one-based indexing (deprecated).

Syntax

```
call mkl_sdiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, c, ldc)
call mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, c, ldc)
call mkl_cdiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, c, ldc)
call mkl_zdiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, c, ldc)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?diasm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the diagonal format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A^T)*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the system of linear equations. If *transa* = 'N' or 'n', then *C* := *alpha*inv(*A*)*B*, If *transa* = 'T' or 't' or 'C' or 'c', then *C* := *alpha*inv(A^T)*B*.
m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
alpha	REAL for mkl_sdiasm. DOUBLE PRECISION for mkl_ddiasm. COMPLEX for mkl_cdiasm. DOUBLE COMPLEX for mkl_zdiasm. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".
val	REAL for mkl_sdiasm.

DOUBLE PRECISION for mkl_ddiasm.

COMPLEX for mkl_cdiasm.

DOUBLE COMPLEX for mkl_zdiasm.

Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.

lval INTEGER. Leading dimension of *val*, *lval*≥*m*. Refer to *lval* description in Diagonal Storage Scheme for more details.

idiag INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*.

> **NOTE**
> All elements of this array must be sorted in increasing order.

Refer to *distance* array description in Diagonal Storage Scheme for more details.

ndiag INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.

b REAL for mkl_sdiasm.

DOUBLE PRECISION for mkl_ddiasm.

COMPLEX for mkl_cdiasm.

DOUBLE COMPLEX for mkl_zdiasm.

Array, size (*ldb*, *n*).

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c REAL for mkl_sdiasm.

DOUBLE PRECISION for mkl_ddiasm.

COMPLEX for mkl_cdiasm.

DOUBLE COMPLEX for mkl_zdiasm.

Array, size *ldc* by *n*.

The leading *m*-by-*n* part of the array *c* contains the matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
 ndiag, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, lval, ndiag
  INTEGER      idiag(*)
  REAL          alpha
  REAL          val(lval,*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
 ndiag, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(lval,*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cdiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
 ndiag, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, lval, ndiag
  INTEGER      idiag(*)
  COMPLEX        alpha
  COMPLEX        val(lval,*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zdiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
 ndiag, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, lval, ndiag
  INTEGER      idiag(*)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(lval,*), b(ldb,*), c(ldc,*)
```

mkl_?skysm

Solves a system of linear matrix equations for a sparse matrix stored using the skyline storage scheme with one-based indexing (deprecated).

Syntax

```
call mkl_sskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
call mkl_dskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
call mkl_cskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
call mkl_zskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_trsmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?skysm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the skyline storage format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A^T)*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the system of linear equations. If *transa* = 'N' or 'n', then *C* := *alpha*inv(*A*)*B*, If *transa* = 'T' or 't' or 'C' or 'c', then *C* := *alpha*inv(A^T)*B*,
m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
alpha	REAL for mkl_sskysm.

DOUBLE PRECISION for mkl_dskysm.

COMPLEX for mkl_cskysm.

DOUBLE COMPLEX for mkl_zskysm.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter *matdescra* (*descra*)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter *matdescra*".

NOTE
General matrices (*matdescra*(1)='G') is not supported.

val

REAL for mkl_sskysm.

DOUBLE PRECISION for mkl_dskysm.

COMPLEX for mkl_cskysm.

DOUBLE COMPLEX for mkl_zskysm.

Array containing the set of elements of the matrix *A* in the skyline profile form.

If *matdescrsa*(2)= 'L', then *val* contains elements from the low triangle of the matrix *A*.

If *matdescrsa*(2)= 'U', then *val* contains elements from the upper triangle of the matrix *A*.

Refer to *values* array description in Skyline Storage Scheme for more details.

pntr

INTEGER. Array of length (*m* + 1) for lower triangle, and (*n* + 1) for upper triangle.

It contains the indices specifying the positions of the first element of the matrix *A* in each row (for the lower triangle) or column (for upper triangle) in the *val* array. Refer to *pointers* array description in Skyline Storage Scheme for more details.

b

REAL for mkl_sskysm.

DOUBLE PRECISION for mkl_dskysm.

COMPLEX for mkl_cskysm.

DOUBLE COMPLEX for mkl_zskysm.

Array, size (*ldb*, *n*).

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c REAL for mkl_sskysm.

DOUBLE PRECISION for mkl_dskysm.

COMPLEX for mkl_cskysm.

DOUBLE COMPLEX for mkl_zskysm.

Array, size *ldc* by *n*.

The leading *m*-by-*n* part of the array *c* contains the matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       pntr(*)
  REAL          alpha
  REAL          val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       pntr(*)
  DOUBLE PRECISION        alpha
  DOUBLE PRECISION        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       pntr(*)
  COMPLEX        alpha
  COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc
  INTEGER       pntr(*)
  DOUBLE COMPLEX        alpha
  DOUBLE COMPLEX        val(*), b(ldb,*), c(ldc,*)
```

mkl_?dnscsr

Convert a sparse matrix in uncompressed representation to the CSR format and vice versa (deprecated).

Syntax

```
call mkl_sdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
call mkl_ddnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
call mkl_cdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
call mkl_zdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use the matrix manipulation routinesfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix *A* between formats: stored as a rectangular array (dense representation) and stored using compressed sparse row (CSR) format (3-array variation).

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

job — INTEGER

Array, contains the following conversion parameters:

- *job*(1): Conversion type.
  - If *job*(1)=0, the rectangular matrix *A* is converted to the CSR format;
  - if *job*(1)=1, the rectangular matrix *A* is restored from the CSR format.
- *job*(2): index base for the rectangular matrix *A*.

- If *job*(2)=0, zero-based indexing for the rectangular matrix *A* is used;
- if *job*(2)=1, one-based indexing for the rectangular matrix *A* is used.

- *job*(3): Index base for the matrix in CSR format.
  - If *job*(3)=0, zero-based indexing for the matrix in CSR format is used;
  - if *job*(3)=1, one-based indexing for the matrix in CSR format is used.
- *job*(4): Portion of matrix.
  - If *job*(4)=0, *adns* is a lower triangular part of matrix *A*;
  - If *job*(4)=1, *adns* is an upper triangular part of matrix *A*;
  - If *job*(4)=2, *adns* is a whole matrix *A*.
- *job*(5)=*nzmax*: maximum number of the non-zero elements allowed if *job*(1)=0.
- *job*(6): job indicator for conversion to CSR format.
  - If *job*(6)=0, only array *ia* is generated for the output storage.
  - If *job*(6)>0, arrays *acsr*, *ia*, *ja* are generated for the output storage.

m — INTEGER. Number of rows of the matrix *A*.

n — INTEGER. Number of columns of the matrix *A*.

adns — (input/output)

REAL for mkl_sdnscsr.

DOUBLE PRECISION for mkl_ddnscsr.

COMPLEX for mkl_cdnscsr.

DOUBLE COMPLEX for mkl_zdnscsr.

If the conversion type is from uncompressed to CSR, on input *adns* contains an uncompressed (dense) representation of matrix *A*.

lda — INTEGER. Specifies the leading dimension of *adns* as declared in the calling (sub)program.

For zero-based indexing of *A*, *lda* must be at least max(1, *n*).

For one-based indexing of *A*, *lda* must be at least max(1, *m*).

acsr — (input/output)

REAL for mkl_sdnscsr.

DOUBLE PRECISION for mkl_ddnscsr.

COMPLEX for mkl_cdnscsr.

DOUBLE COMPLEX for mkl_zdnscsr.

If conversion type is from CSR to uncompressed, on input *acsr* contains the non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja (input/output)INTEGER. If conversion type is from CSR to uncompressed, on input *ja* contains the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia (input/output)INTEGER. Array of length *m* + 1.

If conversion type is from CSR to uncompressed, on input *ia* contains indices of elements in the array *acsr*, such that *ia*(*i*) is the index in the array *acsr* of the first non-zero element from the row *i*.

The value of*ia*(*m* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

Output Parameters

adns If conversion type is from CSR to uncompressed, on output *adns* contains the uncompressed (dense) representation of matrix *A*.

acsr, *ja*, *ia* If conversion type is from uncompressed to CSR, on output *acsr*, *ja*, and *ia* contain the compressed sparse row (CSR) format (3-array variation) of matrix *A* (see Sparse Matrix Storage Formats for a description of the storage format).

info INTEGER. Integer info indicator only for restoring the matrix *A* from the CSR format.

If *info*=0, the execution is successful.

If *info*=*i*, the routine is interrupted processing the *i*-th row because there is no space in the arrays *acsr* and *ja* according to the value *nzmax*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
  INTEGER       job(8)
  INTEGER       m, n, lda, info
  INTEGER      ja(*), ia(m+1)
  REAL         adns(*), acsr(*)

SUBROUTINE mkl_ddnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
  INTEGER       job(8)
  INTEGER       m, n, lda, info
  INTEGER      ja(*), ia(m+1)
  DOUBLE PRECISION       adns(*), acsr(*)
```

```
SUBROUTINE mkl_cdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
  INTEGER       job(8)
  INTEGER       m, n, lda, info
  INTEGER      ja(*), ia(m+1)
  COMPLEX      adns(*), acsr(*)

SUBROUTINE mkl_zdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
  INTEGER       job(8)
  INTEGER       m, n, lda, info
  INTEGER      ja(*), ia(m+1)
  DOUBLE COMPLEX      adns(*), acsr(*)
```

mkl_?csrcoo

Converts a sparse matrix in the CSR format to the coordinate format and vice versa (deprecated).

Syntax

```
call mkl_scsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
call mkl_dcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
call mkl_ccsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
call mkl_zcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use the matrix manipulation routinesfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to coordinate format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

job — INTEGER

Array, contains the following conversion parameters:

job(1)

If *job*(1)=0, the matrix in the CSR format is converted to the coordinate format;

if *job*(1)=1, the matrix in the coordinate format is converted to the CSR format.

if *job*(1)=2, the matrix in the coordinate format is converted to the CSR format, and the column indices in CSR representation are sorted in the increasing order within each row.

job(2)

If *job*(2)=0, zero-based indexing for the matrix in CSR format is used;

if *job*(2)=1, one-based indexing for the matrix in CSR format is used.

job(3)

If *job*(3)=0, zero-based indexing for the matrix in coordinate format is used;

if *job*(3)=1, one-based indexing for the matrix in coordinate format is used.

job(5)

job(5)=*nzmax* - maximum number of the non-zero elements allowed if *job*(1)=0.

job(6) - job indicator.

For conversion to the coordinate format:

If *job*(6)=1, only array *rowind* is filled in for the output storage.

If *job*(6)=2, arrays *rowind*, *colind* are filled in for the output storage.

If *job*(6)=3, all arrays *rowind*, *colind*, *acoo* are filled in for the output storage.

For conversion to the CSR format:

If *job*(6)=0, all arrays *acsr*, *ja*, *ia* are filled in for the output storage.

If *job*(6)=1, only array *ia* is filled in for the output storage.

If *job*(6)=2, then it is assumed that the routine already has been called with the *job*(6)=1, and the user allocated the required space for storing the output arrays *acsr* and *ja*.

n — INTEGER. Dimension of the matrix *A*.

nnz — INTEGER. Specifies the number of non-zero elements of the matrix *A* for *job*(1)≠0.

Refer to *nnz* description in Coordinate Format for more details.

acsr — (input/output)

REAL for mkl_scsrcoo.

DOUBLE PRECISION for mkl_dcsrcoo.

COMPLEX for mkl_ccsrcoo.

DOUBLE COMPLEX for mkl_zcsrcoo.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja (input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia (input/output)INTEGER. Array of length *n* + 1, containing indices of elements in the array *acsr*, such that *ia*(*i*) is the index in the array *acsr* of the first non-zero element from the row *i*. The value of the last element *ia*(*n* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

acoo (input/output)

REAL for mkl_scsrcoo.

DOUBLE PRECISION for mkl_dcsrcoo.

COMPLEX for mkl_ccsrcoo.

DOUBLE COMPLEX for mkl_zcsrcoo.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

rowind (input/output)INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in Coordinate Format for more details.

colind (input/output)INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.

Output Parameters

nnz Returns the number of converted elements of the matrix *A* for *job*(1)=0.

info INTEGER. Integer info indicator only for converting the matrix *A* from the CSR format.

If *info*=0, the execution is successful.

If *info*=1, the routine is interrupted because there is no space in the arrays *acoo*, *rowind*, *colind* according to the value *nzmax*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
  INTEGER      job(8)
  INTEGER      n, nnz, info
  INTEGER      ja(*), ia(n+1), rowind(*), colind(*)
  REAL         acsr(*), acoo(*)

SUBROUTINE mkl_dcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
  INTEGER      job(8)
  INTEGER      n, nnz, info
  INTEGER      ja(*), ia(n+1), rowind(*), colind(*)
  DOUBLE PRECISION       acsr(*), acoo(*)

SUBROUTINE mkl_ccsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
  INTEGER      job(8)
  INTEGER      n, nnz, info
  INTEGER      ja(*), ia(n+1), rowind(*), colind(*)
  COMPLEX      acsr(*), acoo(*)

SUBROUTINE mkl_zcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
  INTEGER      job(8)
  INTEGER      n, nnz, info
  INTEGER      ja(*), ia(n+1), rowind(*), colind(*)
  DOUBLE COMPLEX      acsr(*), acoo(*)
```

mkl_?csrbsr

Converts a square sparse matrix in the CSR format to the BSR format and vice versa (deprecated).

Syntax

```
call mkl_scsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
call mkl_dcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
call mkl_ccsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
call mkl_zcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use the matrix manipulation routinesfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a square sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to the block sparse row (BSR) format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

job — INTEGER

Array, contains the following conversion parameters:

job(1)

If *job*(1)=0, the matrix in the CSR format is converted to the BSR format;

if *job*(1)=1, the matrix in the BSR format is converted to the CSR format.

job(2)

If *job*(2)=0, zero-based indexing for the matrix in CSR format is used;

if *job*(2)=1, one-based indexing for the matrix in CSR format is used.

job(3)

If *job*(3)=0, zero-based indexing for the matrix in the BSR format is used;

if *job*(3)=1, one-based indexing for the matrix in the BSR format is used.

job(4) is only used for conversion to CSR format. By default, the converter saves the blocks without checking whether an element is zero or not. If *job*(4)=1, then the converter only saves non-zero elements in blocks.

job(6) - job indicator.

For conversion to the BSR format:

If *job*(6)=0, only arrays *jab*, *iab* are generated for the output storage.

If *job*(6)>0, all output arrays *absr*, *jab*, and *iab* are filled in for the output storage.

If *job*(6)=-1, *iab*(1) returns the number of non-zero blocks.

For conversion to the CSR format:

If *job*(6)=0, only arrays *ja*, *ia* are generated for the output storage.

m — INTEGER. Actual row dimension of the matrix *A* for convert to the BSR format; block row dimension of the matrix *A* for convert to the CSR format.

mblk — INTEGER. Size of the block in the matrix *A*.

ldabsr — INTEGER. Leading dimension of the array *absr* as declared in the calling program. *ldabsr* must be greater than or equal to *mblk*mblk*.

acsr — (input/output)

REAL for mkl_scsrbsr.

DOUBLE PRECISION for mkl_dcsrbsr.

COMPLEX for mkl_ccsrbsr.

DOUBLE COMPLEX for mkl_zcsrbsr.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja (input/output)INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia (input/output)INTEGER. Array of length *m + 1*, containing indices of elements in the array *acsr*, such that *ia(I)* is the index in the array *acsr* of the first non-zero element from the row *I*. The value of the last element*ia(m + 1)* is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

absr (input/output)

REAL for mkl_scsrbsr.

DOUBLE PRECISION for mkl_dcsrbsr.

COMPLEX for mkl_ccsrbsr.

DOUBLE COMPLEX for mkl_zcsrbsr.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *mblk*mblk*. Refer to *values* array description in BSR Format for more details.

jab (input/output)INTEGER. Array containing the column indices for each non-zero block of the matrix *A*.

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in BSR Format for more details.

iab (input/output)INTEGER. Array of length *(m + 1)*, containing indices of blocks in the array *absr*, such that *iab(i)* is the index in the array *absr* of the first non-zero element from the *i*-th row . The value of the last element*iab(m + 1)* is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in BSR Format for more details.

Output Parameters

info INTEGER. Integer info indicator only for converting the matrix *A* from the CSR format.

If *info*=0, the execution is successful.

If *info*=1, it means that *mblk* is equal to 0.

If *info*=2, it means that *ldabsr* is less than *mblk*mblk* and there is no space for all blocks.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
  INTEGER      job(8)
  INTEGER      m, mblk, ldabsr, info
  INTEGER      ja(*), ia(m+1), jab(*), iab(*)
  REAL         acsr(*), absr(ldabsr,*)

SUBROUTINE mkl_dcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
  INTEGER      job(8)
  INTEGER      m, mblk, ldabsr, info
  INTEGER      ja(*), ia(m+1), jab(*), iab(*)
  DOUBLE PRECISION       acsr(*), absr(ldabsr,*)

SUBROUTINE mkl_ccsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
  INTEGER      job(8)
  INTEGER      m, mblk, ldabsr, info
  INTEGER      ja(*), ia(m+1), jab(*), iab(*)
  COMPLEX       acsr(*), absr(ldabsr,*)

SUBROUTINE mkl_zcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
  INTEGER      job(8)
  INTEGER      m, mblk, ldabsr, info
  INTEGER      ja(*), ia(m+1), jab(*), iab(*)
  DOUBLE COMPLEX       acsr(*), absr(ldabsr,*)
```

mkl_?csrcsc

Converts a square sparse matrix in the CSR format to the CSC format and vice versa (deprecated).

Syntax

```
call mkl_scsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
call mkl_dcsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
call mkl_ccsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
call mkl_zcsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use the matrix manipulation routinesfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a square sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to the compressed sparse column (CSC) format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

job INTEGER

Array, contains the following conversion parameters:

job(1)

If *job*(1)=0, the matrix in the CSR format is converted to the CSC format;

if *job*(1)=1, the matrix in the CSC format is converted to the CSR format.

job(2)

If *job*(2)=0, zero-based indexing for the matrix in CSR format is used;

if *job*(2)=1, one-based indexing for the matrix in CSR format is used.

job(3)

If *job*(3)=0, zero-based indexing for the matrix in the CSC format is used;

if *job*(3)=1, one-based indexing for the matrix in the CSC format is used.

job(6) - job indicator.

For conversion to the CSC format:

If *job*(6)=0, only arrays *ja1*, *ia1* are filled in for the output storage.

If *job*(6)≠0, all output arrays *acsc*, *ja1*, and *ia1* are filled in for the output storage.

For conversion to the CSR format:

If *job*(6)=0, only arrays *ja*, *ia* are filled in for the output storage.

If *job*(6)≠0, all output arrays *acsr*, *ja*, and *ia* are filled in for the output storage.

m INTEGER. Dimension of the square matrix *A*.

acsr (input/output)

REAL for mkl_scsrcsc.

DOUBLE PRECISION for mkl_dcsrcsc.

COMPLEX for mkl_ccsrcsc.

DOUBLE COMPLEX for mkl_zcsrcsc.

Array containing non-zero elements of the square matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja (input/output)INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia (input/output)INTEGER. Array of length *m* + 1, containing indices of elements in the array *acsr*, such that *ia*(*i*) is the index in the array *acsr* of the first non-zero element from the row *i*. The value of the last element*ia*(*m* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

acsc (input/output)

REAL for mkl_scsrcsc.

DOUBLE PRECISION for mkl_dcsrcsc.

COMPLEX for mkl_ccsrcsc.

DOUBLE COMPLEX for mkl_zcsrcsc.

Array containing non-zero elements of the square matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja1 (input/output)INTEGER. Array containing the row indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsc*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia1 (input/output)INTEGER. Array of length *m* + 1, containing indices of elements in the array *acsc*, such that *ia1*(*i*) is the index in the array *acsc* of the first non-zero element from the column *i*. The value of the last element *ia1*(*m* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

Output Parameters

info INTEGER. This parameter is not used now.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
  INTEGER      job(8)
  INTEGER      m, info
  INTEGER      ja(*), ia(m+1), ja1(*), ia1(m+1)
  REAL         acsr(*), acsc(*)
```

```
SUBROUTINE mkl_dcsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
  INTEGER      job(8)
  INTEGER      m, info
  INTEGER      ja(*), ia(m+1), ja1(*), ia1(m+1)
  DOUBLE PRECISION       acsr(*), acsc(*)

SUBROUTINE mkl_ccsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
  INTEGER      job(8)
  INTEGER      m, info
  INTEGER      ja(*), ia(m+1), ja1(*), ia1(m+1)
  COMPLEX      acsr(*), acsc(*)

SUBROUTINE mkl_zcsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
  INTEGER      job(8)
  INTEGER      m, info
  INTEGER      ja(*), ia(m+1), ja1(*), ia1(m+1)
  DOUBLE COMPLEX        acsr(*), acsc(*)
```

mkl_?csrdia

Converts a sparse matrix in the CSR format to the diagonal format and vice versa (deprecated).

Syntax

`call mkl_scsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)`

`call mkl_dcsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)`

`call mkl_ccsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)`

`call mkl_zcsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)`

Include Files

- `mkl.fi`

Description

This routine is deprecated. Use the matrix manipulation routinesfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to the diagonal format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

job

INTEGER

Array, contains the following conversion parameters:

job(1)

If *job*(1)=0, the matrix in the CSR format is converted to the diagonal format;

if *job*(1)=1, the matrix in the diagonal format is converted to the CSR format.

job(2)

If *job*(2)=0, zero-based indexing for the matrix in CSR format is used;

if *job*(2)=1, one-based indexing for the matrix in CSR format is used.

job(3)

If *job*(3)=0, zero-based indexing for the matrix in the diagonal format is used;

if *job*(3)=1, one-based indexing for the matrix in the diagonal format is used.

job(6) - job indicator.

For conversion to the diagonal format:

If *job*(6)=0, diagonals are not selected internally, and *acsr_rem*, *ja_rem*, *ia_rem* are not filled in for the output storage.

If *job*(6)=1, diagonals are not selected internally, and *acsr_rem*, *ja_rem*, *ia_rem* are filled in for the output storage.

If *job*(6)=10, diagonals are selected internally, and *acsr_rem*, *ja_rem*, *ia_rem* are not filled in for the output storage.

If *job*(6)=11, diagonals are selected internally, and *csr_rem*, *ja_rem*, *ia_rem* are filled in for the output storage.

For conversion to the CSR format:

If *job*(6)=0, each entry in the array *adia* is checked whether it is zero. Zero entries are not included in the array *acsr*.

If *job*(6)≠0, each entry in the array *adia* is not checked whether it is zero.

m

INTEGER. Dimension of the matrix *A*.

acsr

(input/output)

REAL for mkl_scsrdia.

DOUBLE PRECISION for mkl_dcsrdia.

COMPLEX for mkl_ccsrdia.

DOUBLE COMPLEX for mkl_zcsrdia.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja (input/output)INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia (input/output)INTEGER. Array of length *m* + 1, containing indices of elements in the array *acsr*, such that *ia*(*i*) is the index in the array *acsr* of the first non-zero element from the row *i*. The value of the last element*ia*(*m* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

adia (input/output)

REAL for mkl_scsrdia.

DOUBLE PRECISION for mkl_dcsrdia.

COMPLEX for mkl_ccsrdia.

DOUBLE COMPLEX for mkl_zcsrdia.

Array of size (*ndiag* x *idiag*) containing diagonals of the matrix *A*.

The key point of the storage is that each element in the array *adia* retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom.

ndiag INTEGER.

Specifies the leading dimension of the array *adia* as declared in the calling (sub)program, must be at least max(1, *m*).

distance INTEGER.

Array of length *idiag*, containing the distances between the main diagonal and each non-zero diagonal to be extracted. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

idiag INTEGER.

Number of diagonals to be extracted. For conversion to diagonal format on return this parameter may be modified.

acsr_rem, *ja_rem*, *ia_rem* Remainder of the matrix in the CSR format if it is needed for conversion to the diagonal format.

Output Parameters

info INTEGER. This parameter is not used now.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem,
ia_rem, info)

  INTEGER      job(8)

  INTEGER      m, info, ndiag, idiag

  INTEGER      ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)

  REAL         acsr(*), adia(*), acsr_rem(*)

SUBROUTINE mkl_dcsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem,
ia_rem, info)

  INTEGER      job(8)

  INTEGER      m, info, ndiag, idiag

  INTEGER      ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)

  DOUBLE PRECISION       acsr(*), adia(*), acsr_rem(*)

SUBROUTINE mkl_ccsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem,
ia_rem, info)

  INTEGER      job(8)

  INTEGER      m, info, ndiag, idiag

  INTEGER      ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)

  COMPLEX      acsr(*), adia(*), acsr_rem(*)

SUBROUTINE mkl_zcsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem,
ia_rem, info)

  INTEGER      job(8)

  INTEGER      m, info, ndiag, idiag

  INTEGER      ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)

  DOUBLE COMPLEX       acsr(*), adia(*), acsr_rem(*)
```

mkl_?csrsky

Converts a sparse matrix in CSR format to the skyline format and vice versa (deprecated).

Syntax

```
call mkl_scsrsky(job, m, acsr, ja, ia, asky, pointers, info)

call mkl_dcsrsky(job, m, acsr, ja, ia, asky, pointers, info)

call mkl_ccsrsky(job, m, acsr, ja, ia, asky, pointers, info)

call mkl_zcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use the matrix manipulation routinesfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to the skyline format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

job INTEGER

Array, contains the following conversion parameters:

job(1)

If *job*(1)=0, the matrix in the CSR format is converted to the skyline format;

if *job*(1)=1, the matrix in the skyline format is converted to the CSR format.

job(2)

If *job*(2)=0, zero-based indexing for the matrix in CSR format is used;

if *job*(2)=1, one-based indexing for the matrix in CSR format is used.

job(3)

If *job*(3)=0, zero-based indexing for the matrix in the skyline format is used;

if *job*(3)=1, one-based indexing for the matrix in the skyline format is used.

job(4)

For conversion to the skyline format:

If *job*(4)=0, the upper part of the matrix *A* in the CSR format is converted.

If *job*(4)=1, the lower part of the matrix *A* in the CSR format is converted.

For conversion to the CSR format:

If *job*(4)=0, the matrix is converted to the upper part of the matrix *A* in the CSR format.

If *job*(4)=1, the matrix is converted to the lower part of the matrix *A* in the CSR format.

job(5)

job(5)=*nzmax* - maximum number of the non-zero elements of the matrix *A* if *job*(1)=0.

job(6) - job indicator.

Only for conversion to the skyline format:

If *job*(6)=0, only arrays *pointers* is filled in for the output storage.

If *job*(6)=1, all output arrays *asky* and *pointers* are filled in for the output storage.

m INTEGER. Dimension of the matrix *A*.

acsr (input/output)

REAL for mkl_scsrsky.

DOUBLE PRECISION for mkl_dcsrsky.

COMPLEX for mkl_ccsrsky.

DOUBLE COMPLEX for mkl_zcsrsky.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja (input/output)INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia (input/output)INTEGER. Array of length *m* + 1, containing indices of elements in the array *acsr*, such that *ia*(*i*) is the index in the array *acsr* of the first non-zero element from the row *i*. The value of the last element*ia*(*m* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

asky (input/output)

REAL for mkl_scsrsky.

DOUBLE PRECISION for mkl_dcsrsky.

COMPLEX for mkl_ccsrsky.

DOUBLE COMPLEX for mkl_zcsrsky.

Array, for a lower triangular part of *A* it contains the set of elements from each row starting from the first none-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets. Refer to *values* array description in Skyline Storage Format for more details.

pointers (input/output)INTEGER.

Array with dimension (*m*+1), where *m* is number of rows for lower triangle (columns for upper triangle), *pointers*(*i*) – *pointers*(1) + 1 gives the index of element in the array *asky* that is first non-zero element in row

(column)*i* . The value of *pointers*(*m* + 1) is set to *nnz* + *pointers*(1), where *nnz* is the number of elements in the array *asky*. Refer to *pointers* array description in Skyline Storage Format for more details

Output Parameters

info INTEGER. Integer info indicator only for converting the matrix *A* from the CSR format.

If *info*=0, the execution is successful.

If *info*=1, the routine is interrupted because there is no space in the array *asky* according to the value *nzmax*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsky(job, m, acsr, ja, ia, asky, pointers, info)
  INTEGER      job(8)
  INTEGER      m, info
  INTEGER      ja(*), ia(m+1), pointers(m+1)
  REAL         acsr(*), asky(*)

SUBROUTINE mkl_dcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
  INTEGER      job(8)
  INTEGER      m, info
  INTEGER      ja(*), ia(m+1), pointers(m+1)
  DOUBLE PRECISION       acsr(*), asky(*)

SUBROUTINE mkl_ccsrsky(job, m, acsr, ja, ia, asky, pointers, info)
  INTEGER      job(8)
  INTEGER      m, info
  INTEGER      ja(*), ia(m+1), pointers(m+1)
  COMPLEX      acsr(*), asky(*)

SUBROUTINE mkl_zcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
  INTEGER      job(8)
  INTEGER      m, info
  INTEGER      ja(*), ia(m+1), pointers(m+1)
  DOUBLE COMPLEX       acsr(*), asky(*)
```

mkl_?csradd

Computes the sum of two matrices stored in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
call mkl_scsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
call mkl_dcsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
call mkl_ccsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
call mkl_zcsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_addfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?csradd routine performs a matrix-matrix operation defined as

```
C := A+beta*op(B)
```

where:

A, *B*, *C* are the sparse matrices in the CSR format (3-array variation).

op(*B*) is one of op(*B*) = *B*, or op(*B*) = B^T, or op(*B*) = B^H

beta is a scalar.

The routine works correctly if and only if the column indices in sparse matrix representations of matrices *A* and *B* are arranged in the increasing order for each row. If not, use the parameter *sort* (see below) to reorder column indices and the corresponding elements of the input matrices.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

trans CHARACTER*1. Specifies the operation.

If *trans* = 'N' or 'n', then *C* := *A*+*beta***B*

If *trans* = 'T' or 't', then *C* := *A*+*beta***B*^{T}

If *trans* = 'C' or 'c', then *C* := *A*+*beta***B*^{H}.

request INTEGER.

If *request*=0, the routine performs addition. The memory for the output arrays *ic*, *jc*, *c* must be allocated beforehand.

If *request*=1, the routine only computes the values of the array *ic* of length *m* + 1. The memory for the *ic* array must be allocated beforehand. On exit the value *ic*(*m*+1) – 1 is the actual number of the elements in the arrays *c* and *jc*.

If *request*=2, after the routine is called previously with the parameter *request*=1 and after the output arrays *jc* and *c* are allocated in the calling program with length at least *ic*(*m*+1) – 1, the routine performs addition.

sort INTEGER. Specifies the type of reordering. If this parameter is not set (default), the routine does not perform reordering.

If *sort*=1, the routine arranges the column indices *ja* for each row in the increasing order and reorders the corresponding values of the matrix *A* in the array *a*.

If *sort*=2, the routine arranges the column indices *jb* for each row in the increasing order and reorders the corresponding values of the matrix *B* in the array *b*.

If *sort*=3, the routine performs reordering for both input matrices *A* and *B*.

m INTEGER. Number of rows of the matrix *A*.

n INTEGER. Number of columns of the matrix *A*.

a REAL for mkl_scsradd.

DOUBLE PRECISION for mkl_dcsradd.

COMPLEX for mkl_ccsradd.

DOUBLE COMPLEX for mkl_zcsradd.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia INTEGER. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zero elements of the matrix *A* plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

beta REAL for mkl_scsradd.

DOUBLE PRECISION for mkl_dcsradd.

COMPLEX for mkl_ccsradd.

DOUBLE COMPLEX for mkl_zcsradd.

Specifies the scalar *beta*.

b

REAL for mkl_scsradd.

DOUBLE PRECISION for mkl_dcsradd.

COMPLEX for mkl_ccsradd.

DOUBLE COMPLEX for mkl_zcsradd.

Array containing non-zero elements of the matrix *B*. Its length is equal to the number of non-zero elements in the matrix *B*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

jb

INTEGER. Array containing the column indices for each non-zero element of the matrix *B*. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array *b*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ib

INTEGER. Array of length *m* + 1 when *trans* = 'N' or 'n', or *n* + 1 otherwise.

This array contains indices of elements in the array *b*, such that *ib*(*i*) is the index in the array *b* of the first non-zero element from the row *i*. The value of the last element *ib*(*m* + 1) or *ib*(*n* + 1) is equal to the number of non-zero elements of the matrix *B* plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

nzmax

INTEGER. The length of the arrays *c* and *jc*.

This parameter is used only if *request*=0. The routine stops calculation if the number of elements in the result matrix *C* exceeds the specified value of *nzmax*.

Output Parameters

c

REAL for mkl_scsradd.

DOUBLE PRECISION for mkl_dcsradd.

COMPLEX for mkl_ccsradd.

DOUBLE COMPLEX for mkl_zcsradd.

Array containing non-zero elements of the result matrix *C*. Its length is equal to the number of non-zero elements in the matrix *C*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

jc

INTEGER. Array containing the column indices for each non-zero element of the matrix *C*.

The length of this array is equal to the length of the array *c*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ic INTEGER. Array of length *m* + 1, containing indices of elements in the array *c*, such that *ic*(*i*) is the index in the array *c* of the first non-zero element from the row *i*. The value of the last element *ic*(*m* + 1) is equal to the number of non-zero elements of the matrix *C* plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

info INTEGER.

If *info*=0, the execution is successful.

If *info*=*I*>0, the routine stops calculation in the *I*-th row of the matrix *C* because number of elements in *C* exceeds *nzmax*.

If *info*=-1, the routine calculates only the size of the arrays *c* and *jc* and returns this value plus 1 as the last element of the array *ic*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic,
nzmax, info)
   CHARACTER  trans
   INTEGER       request, sort, m, n, nzmax, info
   INTEGER       ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
   REAL            a(*), b(*), c(*), beta

SUBROUTINE mkl_dcsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic,
nzmax, info)
   CHARACTER  trans
   INTEGER       request, sort, m, n, nzmax, info
   INTEGER       ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
   DOUBLE PRECISION         a(*), b(*), c(*), beta

SUBROUTINE mkl_ccsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic,
nzmax, info)
   CHARACTER  trans
   INTEGER       request, sort, m, n, nzmax, info
   INTEGER       ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
   COMPLEX     a(*), b(*), c(*), beta

SUBROUTINE mkl_zcsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic,
nzmax, info)
   CHARACTER  trans
   INTEGER       request, sort, m, n, nzmax, info
   INTEGER       ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
   DOUBLE COMPLEX       a(*), b(*), c(*), beta
```

mkl_?csrmultcsr

Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
call mkl_scsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
call mkl_dcsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
call mkl_ccsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
call mkl_zcsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_spmmfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?csrmultcsr routine performs a matrix-matrix operation defined as

```
C := op(A)*B
```

where:

A, *B*, *C* are the sparse matrices in the CSR format (3-array variation);

op(*A*) is one of op(*A*) = *A*, or op(*A*) =A^T, or op(*A*) = A^H .

You can use the parameter *sort* to perform or not perform reordering of non-zero entries in input and output sparse matrices. The purpose of reordering is to rearrange non-zero entries in compressed sparse row matrix so that column indices in compressed sparse representation are sorted in the increasing order for each row.

The following table shows correspondence between the value of the parameter *sort* and the type of reordering performed by this routine for each sparse matrix involved:

Value of the parameter *sort*	**Reordering of *A* (arrays** *a*, *ja*, *ia***)**	**Reordering of *B* (arrays** *b*, *ja*, *ib***)**	**Reordering of *C* (arrays** *c*, *jc*, *ic***)**
1	yes	no	yes
2	no	yes	yes
3	yes	yes	yes
4	yes	no	no
5	no	yes	no
6	yes	yes	no
7	no	no	no
arbitrary value not equal to 1, 2,..., 7	no	no	yes

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

trans

CHARACTER*1. Specifies the operation.

If *trans* = 'N' or 'n', then $C := A*B$

If *trans* = 'T' or 't' or 'C' or 'c', then $C := A^{T}*B$.

request

INTEGER.

If *request*=0, the routine performs multiplication, the memory for the output arrays *ic*, *jc*, *c* must be allocated beforehand.

If *request*=1, the routine computes only values of the array *ic* of length *m* + 1, the memory for this array must be allocated beforehand. On exit the value *ic*(*m*+1) – 1 is the actual number of the elements in the arrays *c* and *jc*.

If *request*=2, the routine has been called previously with the parameter *request*=1, the output arrays *jc* and *c* are allocated in the calling program and they are of the length *ic*(*m*+1) – 1 at least.

sort

INTEGER. Specifies whether the routine performs reordering of non-zeros entries in input and/or output sparse matrices (see table above).

m

INTEGER. Number of rows of the matrix *A*.

n

INTEGER. Number of columns of the matrix *A*.

k

INTEGER. Number of columns of the matrix *B*.

a

REAL for mkl_scsrmultcsr.

DOUBLE PRECISION for mkl_dcsrmultcsr.

COMPLEX for mkl_ccsrmultcsr.

DOUBLE COMPLEX for mkl_zcsrmultcsr.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja

INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia

INTEGER. Array of length *m* + 1.

This array contains indices of elements in the array *a*, such that *ia(i)* is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia(m + 1)* is equal to the number of non-zero elements of the matrix *A* plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

b

REAL for mkl_scsrmultcsr.

DOUBLE PRECISION for mkl_dcsrmultcsr.

COMPLEX for mkl_ccsrmultcsr.

DOUBLE COMPLEX for mkl_zcsrmultcsr.

Array containing non-zero elements of the matrix *B*. Its length is equal to the number of non-zero elements in the matrix *B*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

jb

INTEGER. Array containing the column indices for each non-zero element of the matrix *B*. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array *b*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ib

INTEGER. Array of length *n + 1* when *trans = 'N'* or *'n'*, or *m + 1* otherwise.

This array contains indices of elements in the array *b*, such that *ib(i)* is the index in the array *b* of the first non-zero element from the row *i*. The value of the last element *ib(n + 1)* or *ib(m + 1)* is equal to the number of non-zero elements of the matrix *B* plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

nzmax

INTEGER. The length of the arrays *c* and *jc*.

This parameter is used only if *request*=0. The routine stops calculation if the number of elements in the result matrix *C* exceeds the specified value of *nzmax*.

Output Parameters

c

REAL for mkl_scsrmultcsr.

DOUBLE PRECISION for mkl_dcsrmultcsr.

COMPLEX for mkl_ccsrmultcsr.

DOUBLE COMPLEX for mkl_zcsrmultcsr.

Array containing non-zero elements of the result matrix *C*. Its length is equal to the number of non-zero elements in the matrix *C*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

jc

INTEGER. Array containing the column indices for each non-zero element of the matrix *C*.

The length of this array is equal to the length of the array *c*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ic INTEGER. Array of length *m* + 1 when *trans* = 'N' or 'n', or *n* + 1 otherwise.

This array contains indices of elements in the array *c*, such that *ic*(*i*) is the index in the array *c* of the first non-zero element from the row *i*. The value of the last element *ic*(*m* + 1) or *ic*(*n* + 1) is equal to the number of non-zero elements of the matrix *C* plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

info INTEGER.

If *info*=0, the execution is successful.

If *info*=*I*>0, the routine stops calculation in the *I*-th row of the matrix *C* because number of elements in *C* exceeds *nzmax*.

If *info*=-1, the routine calculates only the size of the arrays *c* and *jc* and returns this value plus 1 as the last element of the array *ic*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic,
nzmax, info)

   CHARACTER*1  trans

   INTEGER       request, sort, m, n, k, nzmax, info

   INTEGER       ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)

   REAL            a(*), b(*), c(*)

SUBROUTINE mkl_dcsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic,
nzmax, info)

   CHARACTER*1  trans

   INTEGER       request, sort, m, n, k, nzmax, info

   INTEGER       ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)

   DOUBLE PRECISION            a(*), b(*), c(*)

SUBROUTINE mkl_ccsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic,
nzmax, info)

   CHARACTER*1  trans

   INTEGER       request, sort, m, n, k, nzmax, info

   INTEGER       ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)

   COMPLEX     a(*), b(*), c(*)
```

```
SUBROUTINE mkl_zcsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic,
nzmax, info)
   CHARACTER*1  trans
   INTEGER       request, sort, m, n, k, nzmax, info
   INTEGER       ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
   DOUBLE COMPLEX       a(*), b(*), c(*)
```

mkl_?csrmultd

Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix (deprecated).

Syntax

```
call mkl_scsrmultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
call mkl_dcsrmultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
call mkl_ccsrmultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
call mkl_zcsrmultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

Include Files

- mkl.fi

Description

This routine is deprecated. Use mkl_sparse_?_spmmdfrom the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The mkl_?csrmultd routine performs a matrix-matrix operation defined as

```
C := op(A)*B
```

where:

A, *B* are the sparse matrices in the CSR format (3-array variation), *C* is dense matrix;

op(*A*) is one of op(*A*) = *A*, or op(*A*) = A^T, or op(*A*) = A^H .

The routine works correctly if and only if the column indices in sparse matrix representations of matrices *A* and *B* are arranged in the increasing order for each row.

> **NOTE**
> This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

trans CHARACTER*1. Specifies the operation.

If *trans* = 'N' or 'n', then *C* := *A***B*

If *trans* = 'T' or 't' or 'C' or 'c', then *C* := A^T**B*.

m INTEGER. Number of rows of the matrix *A*.

n INTEGER. Number of columns of the matrix *A*.

k INTEGER. Number of columns of the matrix *B*.

a REAL for mkl_scsrmultd.

DOUBLE PRECISION for mkl_dcsrmultd.

COMPLEX for mkl_ccsrmultd.

DOUBLE COMPLEX for mkl_zcsrmultd.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

ja INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ia INTEGER. Array of length *m* + 1 when *trans* = 'N' or 'n', or *n* + 1 otherwise.

This array contains indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) or *ia*(*n* + 1) is equal to the number of non-zero elements of the matrix *A* plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

b REAL for mkl_scsrmultd.

DOUBLE PRECISION for mkl_dcsrmultd.

COMPLEX for mkl_ccsrmultd.

DOUBLE COMPLEX for mkl_zcsrmultd.

Array containing non-zero elements of the matrix *B*. Its length is equal to the number of non-zero elements in the matrix *B*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.

jb INTEGER. Array containing the column indices for each non-zero element of the matrix *B*. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array *b*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.

ib INTEGER. Array of length *m* + 1.

This array contains indices of elements in the array *b*, such that *ib(i)* is the index in the array *b* of the first non-zero element from the row *i*. The value of the last element *ib(m + 1)* is equal to the number of non-zero elements of the matrix *B* plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.

Output Parameters

c

REAL for mkl_scsrmultd.

DOUBLE PRECISION for mkl_dcsrmultd.

COMPLEX for mkl_ccsrmultd.

DOUBLE COMPLEX for mkl_zcsrmultd.

Array containing non-zero elements of the result matrix *C*.

ldc

INTEGER. Specifies the leading dimension of the dense matrix *C* as declared in the calling (sub)program. Must be at least max(*m*, 1) when *trans* = 'N' or 'n', or max(1, *n*) otherwise.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrmultd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
   CHARACTER*1  trans
   INTEGER       m, n, k, ldc
   INTEGER       ja(*), jb(*), ia(*), ib(*)
   REAL            a(*), b(*), c(ldc, *)

SUBROUTINE mkl_dcsrmultd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
   CHARACTER*1  trans
   INTEGER       m, n, k, ldc
   INTEGER       ja(*), jb(*), ia(*), ib(*)
   DOUBLE PRECISION         a(*), b(*), c(ldc, *)

SUBROUTINE mkl_ccsrmultd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
   CHARACTER*1  trans
   INTEGER       m, n, k, ldc
   INTEGER       ja(*), jb(*), ia(*), ib(*)
   COMPLEX     a(*), b(*), c(ldc, *)
```

```
SUBROUTINE mkl_zcsrmultd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
   CHARACTER*1  trans
   INTEGER       m, n, k, ldc
   INTEGER       ja(*), jb(*), ia(*), ib(*)
   DOUBLE COMPLEX      a(*), b(*), c(ldc, *)
```

Sparse QR Routines

Sparse QR routines and their data types

Routine or function group	Data types	Description
mkl_sparse_set_qr_hint		Enables a pivot strategy for an ill-conditioned matrix.
mkl_sparse_?_qr	s,d	Calculates the solution of a sparse system of linear equations using QR factorization.
mkl_sparse_qr_reorder		Performs reordering and symbolic analysis of the matrix *A*.
mkl_sparse_?_qr_factorize	s,d	Performs numerical factorization of the matrix *A*.
mkl_sparse_?_qr_solve	s,d	Solves the system *A*x = b* using QR factorization of the matrix *A*.
mkl_sparse_?_qr_qmult	s,d	Performs *x := Q^(-1)*b*.
mkl_sparse_?_qr_rsolve	s,d	Performs *x := R^(-1)*b*.

NOTE The underdetermined systems of equations are not supported. The number of columns should be less or equal to the number or rows.

For more information about the workflow of sparse QR functionality, refer to oneMKL Sparse QR solver. Multifrontal Sparse QR Factorization Method for Solving a Sparse System of Linear Equations.

mkl_sparse_set_qr_hint

Define the pivot strategy for further calls of `mkl_sparse_?_qr`.

Syntax

```
stat = mkl_sparse_set_qr_hint ( A, hint )
```

Include Files

- `mkl_sparse_qr.f90`

Description

You can use this routine to enable a pivot strategy in the case of an ill-conditioned matrix.

Input Parameters

A — `SPARSE_MATRIX_T`

Handle containing a sparse matrix in an internal data structure.

hint — `C_INT`

Value specifying whether to use pivoting.

NOTE The only value currently supported is SPARSE_QR_WITH_PIVOTS, which enables the use of a pivot strategy for an ill-conditioned matrix.

Output Parameters

stat INTEGER

Value indicating whether the operation was successful, and if not, why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_qr

Computes the QR decomposition for the matrix of a sparse linear system and calculates the solution.

Syntax

```
stat = mkl_sparse_d_qr (operation, A, descr, layout, *columns, x, *ldx, b, ldb)
stat = mkl_sparse_s_qr (operation, A, descr, layout, x, columns, ldx, b, ldb)
```

Include Files

- mkl_sparse_qr.f90

Description

The mkl_sparse_?_qr routine computes the QR decomposition for the matrix of a sparse linear system $A*x = b$, so that $A = Q*R$ where Q is the orthogonal matrix and R is upper triangular, and calculates the solution.

NOTE
Currently, mkl_sparse_?_qr supports only square and overdetermined systems. For underdetermined systems you can manually transpose the system matrix and use QR decomposition for A^T to get the minimum-norm solution for the original underdetermined system.

NOTE Currently, mkl_sparse_?_qr supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Input Parameters

operation C_INT

Specifies the operation to perform.

> **NOTE** Currently, the only suppored value is SPARSE_OPERATION_NON_TRANSPOSE (non-transpose case; that is, *A*x* = *b* is solved).

A SPARSE_MATRIX_T

Handle containing a sparse matrix in an internal data structure.

descr MATRIX_DESCR

Structure specifying sparse matrix properties. Only the parameters listed here are currently supported.

type Specifies the type of sparse matrix.

> **NOTE** Currently, the only supported value is SPARSE_MATRIX_TYPE_GENERAL (the matrix is processed as-is).

layout C_INT

Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements uses column-major layout.
SPARSE_LAYOUT_ROW_MAJOR	Storage of elements uses row-major layout.

x C_FLOAT for mkl_sparse_s_qr; C_DOUBLE for mkl_sparse_d_qr

Array with a size of at least *rows***cols*:

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR
rows (number of rows in *x*)	*ldx*	Number of columns in *A*
cols (number of columns in *x*)	*columns*	*ldx*

columns C_INT

Number of columns in matrix *b*.

ldx C_INT

Specifies the leading dimension of matrix *x*.

b C_FLOAT for mkl_sparse_s_qr; C_DOUBLE for mkl_sparse_d_qr

Array with a size of at least *rows***cols*:

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR
rows (number of rows in *b*)	*ldb*	Number of columns in *A*
cols (number of columns in *b*)	*columns*	*ldb*

ldb

C_INT

Specifies the leading dimension of matrix *b*.

Output Parameters

x

C_FLOAT for mkl_sparse_s_qr; C_DOUBLE for mkl_sparse_d_qr

Overwritten by the updated matrix *y*.

stat

INTEGER

Value indicating whether the operation was successful, and if not, why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_qr_reorder

Reordering step of SPARSE QR solver.

Syntax

```
stat = mkl_sparse_qr_reorder ( A, descr )
```

Include Files

- mkl_sparse_qr.f90

Description

The mkl_sparse_qr_reorder routine performs ordering and symbolic analysis of matrix *A*.

NOTE Currently, `mkl_sparse_qr_reorder` supports only general structure and CSR format for the input matrix.

Input Parameters

A `SPARSE_MATRIX_T`

Handle containing a sparse matrix in an internal data structure.

descr `MATRIX_DESCR`

Structure specifying sparse matrix properties. Only the parameters listed here are currently supported.

Output Parameters

stat `INTEGER`

Value indicating whether the operation was successful, and if not, why:

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.
`SPARSE_STATUS_EXECUTION_FAILED`	Execution failed.
`SPARSE_STATUS_INTERNAL_ERROR`	An error in algorithm implementation occurred.
`SPARSE_STATUS_NOT_SUPPORTED`	The requested operation is not supported.

mkl_sparse_?_qr_factorize

Factorization step of the SPARSE QR solver.

Syntax

```
stat = mkl_sparse_d_qr_factorize (A, alt_values)
stat = mkl_sparse_s_qr_factorize (A, alt_values)
```

Include Files

- `mkl_sparse_qr.f90`

Description

The `mkl_sparse_?_qr_factorize` routine performs numerical factorization of matrix *A*. Prior to calling this routine, the `mkl_sparse_?_qr_reorder` routine must be called for the matrix handle *A*. For more information about the workflow of sparse QR functionality, refer to oneMKL Sparse QR solver. Multifrontal Sparse QR Factorization Method for Solving a Sparse System of Linear Equations.

NOTE Currently, `mkl_sparse_?_qr_factorize` supports only CSR format for the input matrix.

Input Parameters

A — SPARSE_MATRIX_T

Handle containing a sparse matrix in an internal data structure.

alt_values — C_FLOAT for mkl_sparse_s_qr_factorize; C_DOUBLE for mkl_sparse_d_qr_factorize

Array with alternative values. Must be the size of the non-zeroes in the initial input matrix. When passed to the routine, these values will be used during the factorization step instead of the values stored in handle *A*.

Output Parameters

stat — INTEGER

Value indicating whether the operation was successful, and if not, why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_qr_solve

Solving step of the SPARSE QR solver.

Syntax

```
stat = mkl_sparse_d_qr_solve (operation, A, alt_values, layout, x, columns, ldx, b, ldb)
stat = mkl_sparse_s_qr_solve (operation, A, alt_values, layout, x, columns, ldx, b, ldb)
```

Include Files

- mkl_sparse_qr.f90

Description

The mkl_sparse_?_qr_solve routine computes the solution of sparse systems of linear equations *A*x* = *b*. Prior to calling this routine, the mkl_sparse_?_qr_factorize routine must be called for the matrix handle *A*. For more information about the workflow of sparse QR functionality, refer to oneMKL Sparse QR solver. Multifrontal Sparse QR Factorization Method for Solving a Sparse System of Linear Equations.

NOTE
Currently, `mkl_sparse_?_qr_solve` supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Alternative values are not supported and must be set to NULL.

Input Parameters

operation

`C_INT`

Specifies the operation to perform.

NOTE Currently, the only supported value is `SPARSE_OPERATION_NON_TRANSPOSE` (non-transpose case; that is, *A*x = b* is solved).

A

`SPARSE_MATRIX_T`

Handle containing a sparse matrix in an internal data structure.

alt_values

`C_FLOAT` for `mkl_sparse_s_qr_solve`; `C_DOUBLE` for `mkl_sparse_d_qr_solve`

Reserved for future use.

layout

`C_INT`

Describes the storage scheme for the dense matrix:

`SPARSE_LAYOUT_COLUMN_MAJOR`	Storage of elements uses column-major layout.
`SPARSE_LAYOUT_ROW_MAJOR`	Storage of elements uses row-major layout.

x

`C_FLOAT` for `mkl_sparse_s_qr`; `C_DOUBLE` for `mkl_sparse_d_qr`

Array with a size of at least *rows*cols*:

	layout = `SPARSE_LAYOUT_COLUMN_MAJOR`	*layout* = `SPARSE_LAYOUT_ROW_MAJOR`
rows (number of rows in *x*)	*ldx*	Number of columns in *A*
cols (number of columns in *x*)	*columns*	*ldx*

columns

`C_INT`

Number of columns in matrix *b*.

ldx

`C_INT`

Specifies the leading dimension of matrix *x*.

b

`C_FLOAT for mkl_sparse_s_qr; C_DOUBLE for mkl_sparse_d_qr`

Array with a size of at least *rows*cols*:

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR
rows (number of rows in *b*)	*ldb*	Number of columns in *A*
cols (number of columns in *b*)	*columns*	*ldb*

ldb C_INT

Specifies the leading dimension of matrix *b*.

Output Parameters

x C_FLOAT for mkl_sparse_s_qr; C_DOUBLE for mkl_sparse_d_qr

Contains the solution of system *A*x = b*.

stat INTEGER

Value indicating whether the operation was successful, and if not, why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_qr_qmult

First stage of the solving step of the SPARSE QR solver.

Syntax

```
stat = mkl_sparse_d_qr_qmult (operation, A, layout, x, columns, ldx, b, ldb)
stat = mkl_sparse_s_qr_qmult (operation, A, layout, x, columns, ldx, b, ldb)
```

Include Files

- mkl_sparse_qr.f90

Description

The mkl_sparse_?_qr_qmult routine computes multiplication of inversed matrix *Q* and right-hand side matrix *b*. This routine can be used to perform the solving step in two separate calls as an alternative to a single call of mkl_sparse_?_qr_solve.

NOTE Currently, `mkl_sparse_?_qr_qmult` supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Input Parameters

operation

`C_INT`

Specifies the operation to perform.

NOTE Currently, the only supported value is `SPARSE_OPERATION_NON_TRANSPOSE` (non-transpose case; that is, *A*x = b* is solved).

A

`SPARSE_MATRIX_T`

Handle containing a sparse matrix in an internal data structure.

layout

`C_INT`

Describes the storage scheme for the dense matrix:

`SPARSE_LAYOUT_COLUMN_MAJOR`	Storage of elements uses column-major layout.
`SPARSE_LAYOUT_ROW_MAJOR`	Storage of elements uses row-major layout.

x

`C_FLOAT` for `mkl_sparse_s_qr`; `C_DOUBLE` for `mkl_sparse_d_qr`

Array with a size of at least *rows*cols*:

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR
rows (number of rows in *x*)	*ldx*	Number of columns in *A*
cols (number of columns in *x*)	*columns*	*ldx*

columns

`C_INT`

Number of columns in matrix *b*.

ldx

`C_INT`

Specifies the leading dimension of matrix *x*.

b

`C_FLOAT for mkl_sparse_s_qr; C_DOUBLE for mkl_sparse_d_qr`

Array with a size of at least *rows*cols*:

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR

rows (number of rows in *b*)	*ldb*	Number of columns in *A*
cols (number of columns in *b*)	*columns*	*ldb*

ldb C_INT

Specifies the leading dimension of matrix *b*.

Output Parameters

x C_FLOAT for mkl_sparse_s_qr; C_DOUBLE for mkl_sparse_d_qr

Overwritten by the updated matrix $x = Q^{-1}*b$.

stat INTEGER

Value indicating whether the operation was successful, and if not, why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_qr_rsolve

Second stage of the solving step of the SPARSE QR solver.

Syntax

```
stat = mkl_sparse_d_qr_rsolve (operation, A, layout, x, columns, ldx, b, ldb)
stat = mkl_sparse_s_qr_rsolve (operation, A, layout, x, columns, ldx, b, ldb)
```

Include Files

- mkl_sparse_qr.f90

Description

The mkl_sparse_?_qr_rsolve routine computes the solution of $A*x = b$.

NOTE Currently, `mkl_sparse_?_qr_rsolve` supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Input Parameters

operation

`C_INT`

Specifies the operation to perform.

NOTE Currently, the only supported value is `SPARSE_OPERATION_NON_TRANSPOSE` (non-transpose case; that is, *A*x = b* is solved).

A

`SPARSE_MATRIX_T`

Handle containing a sparse matrix in an internal data structure.

layout

`C_INT`

Describes the storage scheme for the dense matrix:

`SPARSE_LAYOUT_COLUMN_MAJOR`	Storage of elements uses column-major layout.
`SPARSE_LAYOUT_ROW_MAJOR`	Storage of elements uses row-major layout.

x

`C_FLOAT` for `mkl_sparse_s_qr`; `C_DOUBLE` for `mkl_sparse_d_qr`

Array with a size of at least *rows*cols*:

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR
rows (number of rows in *x*)	*ldx*	Number of columns in *A*
cols (number of columns in *x*)	*columns*	*ldx*

columns

`C_INT`

Number of columns in matrix *b*.

ldx

`C_INT`

Specifies the leading dimension of matrix *x*.

b

`C_FLOAT for mkl_sparse_s_qr; C_DOUBLE for mkl_sparse_d_qr`

Array with a size of at least *rows*cols*:

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR

rows (number of rows in `b`)	`ldb`	Number of columns in `A`
cols (number of columns in `b`)	`columns`	`ldb`

`ldb`

`C_INT`

Specifies the leading dimension of matrix `b`.

Output Parameters

`x`

`C_FLOAT` for `mkl_sparse_s_qr`; `C_DOUBLE` for `mkl_sparse_d_qr`

Contains the solution of the triangular system *R*x = b*.

`stat`

`INTEGER`

Value indicating whether the operation was successful, and if not, why:

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.
`SPARSE_STATUS_EXECUTION_FAILED`	Execution failed.
`SPARSE_STATUS_INTERNAL_ERROR`	An error in algorithm implementation occurred.
`SPARSE_STATUS_NOT_SUPPORTED`	The requested operation is not supported.

Inspector-executor Sparse BLAS Routines

The inspector-executor API for Sparse BLAS divides operations into two stages: analysis and execution. During the initial analysis stage, the API inspects the matrix sparsity pattern and applies matrix structure changes. In the execution stage, subsequent routine calls reuse this information in order to improve performance.

The inspector-executor API supports key Sparse BLAS operations for iterative sparse solvers:

- Sparse matrix-vector multiplication
- Sparse matrix-matrix multiplication with a sparse or dense result
- Solution of triangular systems
- Sparse matrix addition

Naming Conventions in Inspector-Executor Sparse BLAS Routines

The Inspector-Executor Sparse BLAS API routine names use the following convention:

```
mkl_sparse_[<character>_]<operation>[_<format>]
```

The <*character*> field indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

The data type is included in the name only if the function accepts dense matrix or scalar floating point parameters.

The <*operation*> field indicates the type of operation:

create	create matrix handle
copy	create a copy of matrix handle
convert	convert matrix between sparse formats
export	export matrix from internal representation to CSR or BSR format
destroy	frees memory allocated for matrix handle
set_<*op*>_hint	provide information about number of upcoming compute operations and operation type for optimization purposes, where <*op*> is mv, sv, mm, sm, dotmv, symgs, or memory
optimize	analyze the matrix using hints and store optimization information in matrix handle
mv	compute sparse matrix-vector product
mm	compute sparse matrix by dense matrix product (batch mv)
set_value	change a value in a matrix
spmm/spmmd	compute sparse matrix by sparse matrix product and store the result as a sparse/dense matrix
trsv	solve a triangular system
trsm	solve a triangular system with multiple right-hand sides
add	compute sum of two sparse matrices
symgs	compute a symmetric Gauss-Zeidel preconditioner
symgs_mv	compute a symmetric Gauss-Zeidel preconditioner with a final matrix-vector multiplication
sorv	computes forward, backward sweeps or symmetric successive over-relaxation preconditioner
sypr	compute the symmetric or Hermitian product of sparse matrices and store the result as a sparse matrix
syprd	compute the symmetric or Hermitian product of sparse and dense matrices and store the result as a dense matrix

`syrk`	compute the product of sparse matrix with its transposed matrix and store the result as a sparse matrix
`syrkd`	compute the product of sparse matrix with its transposed matrix and store the result as a dense matrix
`order`	perform ordering of column indexes of the matrix in CSR format
`dotmv`	compute a sparse matrix-vector product with dot product

The `<format>` field indicates the sparse matrix storage format:

`coo`	coordinate format
`bsr`	block sparse row format plus variations. Fill out either rows_start and rows_end (for 4-arrays representation) or rowIndex array (for 3-array BSR/CSR).
`csr`	compressed sparse row format plus variations. Fill out either `rows_start` and `rows_end` (for 4-arrays representation) or `rowIndex` array (for 3-array BSR/CSR).
`csc`	compressed sparse column format plus variations. Fill out either `cols_start` and `cols_end` (for 4-arrays representation) or `colIndex` array (for 3 array CSC).

The format is included in the function name only if the function parameters include an explicit sparse matrix in one of the conventional sparse matrix formats.

Sparse Matrix Storage Formats for Inspector-executor Sparse BLAS Routines

Inspector-executor Sparse BLAS routines support four conventional sparse matrix storage formats:

- compressed sparse row format (CSR) plus variations
- compressed sparse column format (CSC) plus variations
- coordinate format (COO)
- block sparse row format (BSR) plus variations

Computational routines operate on a matrix handle that stores a matrix in CSR or BSR formats. Other formats should be converted to CSR or BSR format before calling any computational routines. For more information see Sparse Matrix Storage Formats.

Supported Inspector-executor Sparse BLAS Operations

The Inspector-executor Sparse BLAS API can perform several operations involving sparse matrices. These notations are used in the description of the operations:

- *A*, *G*, *V* are sparse matrices
- *B* and *C* are dense matrices
- *x* and *y* are dense vectors
- *alpha* and *beta* are scalars

`op(A)` represents a possible transposition of matrix *A*

`op(A)` = *A*
`op(A)` = A^T - transpose of *A*
`op(A)` = A^H - conjugate transpose of *A*

`op(A)`$^{-1}$ denotes the inverse of `op(A)`.

The Inspector-executor Sparse BLAS routines support the following operations:

- computing the vector product between a sparse matrix and a dense vector:

```
y := alpha*op(A)*x + beta*y
```

- solving a single triangular system:

```
y := alpha*inv(op(A))*x
```

- computing a product between a sparse matrix and a dense matrix:

```
C := alpha*op(A)*B + beta*C
```

- computing a product between sparse matrices with a sparse result:

```
V := alpha*op(A)*op(G)
```

- computing a product between sparse matrices with a dense result:

```
C := alpha*op(A)*op(G)
```

- computing a sum of sparse matrices with a sparse result:

```
V := alpha*op(A) + G
```

- solving a sparse triangular system with multiple right-hand sides:

```
C := alpha*inv(op(A))*B
```

Two-stage Algorithm in Inspector-Executor Sparse BLAS Routines

You can use a two-stage algorithm in Inspector-executor Sparse BLAS routines which produce a sparse matrix. The applicable routines are:

- mkl_sparse_sp2m (BSR/CSR/CSC formats)
- mkl_sparse_sypr (CSR format)

The two-stage algorithm allows you to split computations into stages. The main purpose of the splitting is to provide an estimate for the memory required for the output prior to allocating the largest part of the memory (for the indices and values of the non-zero elements). Additionally, the two-stage approach extends the functionality and allows more complex usage models.

NOTE The multistage approach currently does not allow you to allocate memory for the output matrix outside oneMKL.

In the two-stage algorithm:

1. The first stage allocates data which is necessary for the memory estimation (arrays *rows_start*/*rows_end* or *cols_start*/*cols_end* depending on the format, (see Sparse Matrix Storage Formats) and computes the number of entries or the full structure of the matrix.

   NOTE The format of the output is decided internally but can be checked using the export functionality `mkl_sparse_?_export_<format>`.

2. The second stage allocates data and computes column or row indices (depending on the format) of non-zero elements and/or values of the output matrix.

Specifying the stage for execution is supported through the *sparse_request_t* parameter in the API with the following options:

Values for *sparse_request_t* parameter

Value	Description
SPARSE_STAGE_NNZ_COUNT	Allocates and computes only the *rows_start*/*rows_end* (CSR/BSR format) or *cols_start*/*cols_end* (CSC format) arrays for the output matrix. After this stage, by calling `mkl_sparse_?_export_<format>`, you can obtain the number of non-zeros in the output matrix and calculate the amount of memory required for the output matrix.

Value	Description
SPARSE_STAGE_FINALIZE_MULT_NO_VAL	Allocates and computes row/column indices provided that *rows_start*/*rows_end* or *cols_start*/*cols_end* have already been computed in a prior call with the request *SPARSE_STAGE_NNZ_COUNT*. The values of the output matrix are not computed.
SPARSE_STAGE_FINALIZE_MULT	Depending on the state of the output matrix *C* on entry to the routine, this stage does one of the following: • Allocates and computes row/column indices and values of nonzero elements, if only *rows_start*/*rows_end* or *cols_start*/*cols_end* are present • allocates and computes values of nonzero elements, if *rows_start*/*rows_end* or *cols_start*/*cols_end* and row/column indices of non-zero elements are present
SPARSE_STAGE_FULL_MULT_NO_VAL	Allocates and computes the output matrix structure in a single step. The values of the output matrix are not computed.
SPARSE_STAGE_FULL_MULT	Allocates and computes the entire output matrix (structure and values) in a single step.

The example below shows how you can use the two-stage approach for estimating the memory requirements for the output matrix in CSR format:

First stage (`sparse_request_t` = *SPARSE_STAGE_NNZ_COUNT*)

1. The routine mkl_sparse_sp2m is called with the request parameter *SPARSE_STAGE_NNZ_COUNT*.
2. The arrays `rows_start` and `rows_end` are exported using the mkl_sparse_x_export_csr routine.
3. These arrays are used to calculate the number of non-zeros (nnz) of the resulting output matrix.

Note that by the end of the first stage, the arrays associated with column indices and values of the output matrix have not been allocated or computed yet.

```
sparse_matrix_t csrC = NULL;
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB, csrB, SPARSE_STAGE_NNZ_COUNT, &csrC);

/* optional calculation of nnz in the output matrix for getting a memory estimate */

status = mkl_sparse_?_export_csr (csrC, &indexing, &nrows, &ncols, &rows_start, &rows_end,
&col_indx, &values);

MKL_INT nnz = rows_end[nrows-1] - rows_start[0];
```

Second stage (`sparse_request_t` = *SPARSE_STAGE_FINALIZE_MULT*)

This stage allocates and computes the remaining output arrays (associated with column indices and values of output matrix entries) and completes the matrix-matrix multiplication.

```
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB, csrB, SPARSE_STAGE_FINALIZE_MULT,
&csrC);
```

When the two-stage approach is not needed, you can perform both stages in a single call:

Single stage operation (`sparse_request_t` = *SPARSE_STAGE_FULL_MULT*)

```
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB, csrB, SPARSE_STAGE_FULL_MULT, &csrC);
```

Matrix Manipulation Routines

The Matrix Manipulation Routines table lists the matrix manipulation routines and the data types associated with them.

Matrix Manipulation Routines and Their Data Types

Routine or Function Group	Data Types	Description
mkl_sparse_?_create_csr	s, d, c, z	Creates a handle for a CSR-format matrix.
mkl_sparse_?_create_csc	s, d, c, z	Creates a handle for a CSC format matrix.
mkl_sparse_?_create_coo	s, d, c, z	Creates a handle for a matrix in COO format.
mkl_sparse_?_create_bsr	s, d, c, z	Creates a handle for a matrix in BSR format.
mkl_sparse_copy	NA	Creates a copy of a matrix handle.
mkl_sparse_destroy	NA	Frees memory allocated for matrix handle.
mkl_sparse_convert_csr	NA	Converts internal matrix representation to CSR format.
mkl_sparse_convert_bsr	NA	Converts internal matrix representation to BSR format or changes BSR block size.
mkl_sparse_?_export_csr	s, d, c, z	Exports CSR matrix from internal representation.
mkl_sparse_?_export_csc	s, d, c, z	Exports CSC matrix from internal representation.
mkl_sparse_?_export_bsr	s, d, c, z	Exports BSR matrix from internal representation.
mkl_sparse_?_set_value	s, d, c, z	Changes a single value of matrix in internal representation.
mkl_sparse_?_update_values	s, d, c, z	Changes all or selected matrix values in internal representation.
mkl_sparse_order	NA	Performs ordering of column indexes of the matrix in CSR format.

mkl_sparse_?_create_csr

Creates a handle for a CSR-format matrix.

Syntax

```
stat = mkl_sparse_s_create_csr (A, indexing, rows, cols, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_d_create_csr (A, indexing, rows, cols, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_c_create_csr (A, indexing, rows, cols, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_z_create_csr (A, indexing, rows, cols, rows_start, rows_end, col_indx, values)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_create_csr routine creates a handle for an *m*-by-*k* matrix *A* in CSR format.

> **NOTE**
> The input arrays provided are left unchanged except for the call to mkl_sparse_order, which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use mkl_sparse_copy.

Input Parameters

indexing

sparse_index_base_t.

Indicates how input arrays are indexed.

SPARSE_INDEX_BASE_ZERO	Zero-based (C-style) indexing: indices start at 0.
SPARSE_INDEX_BASE_ONE	One-based (Fortran-style) indexing: indices start at 1.

rows

C_INT.

Number of rows of matrix *A*.

cols

C_INT.

Number of columns of matrix *A*.

rows_start

C_INT.

Array of length at least *rows*. This array contains row indices, such that *rows_start*(*i*) - *indexing* is the first index of row *i* in the arrays *values* and *col_indx*. The value of *indexing* is 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerB* array description in CSR Format for more details.

rows_end

C_INT.

Array of at least length *rows*. This array contains row indices, such that *rows_end*(*i*) - *indexing* – 1 is the last index of row *i* in the arrays *values* and *col_indx*. The value of *indexing* is 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerE* array description in CSR Format for more details.

col_indx

C_INT .

For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix *A*. For zero-based indexing, array containing the column indices for each non-zero element of the matrix *A*. Its length is at least *rows_end*(*rows* – 1) - *indexing*.

The value of *indexing* is 0 for zero-based indexing and 1 for one-based indexing.

values

C_FLOAT for mkl_sparse_s_create_csr

C_DOUBLE for mkl_sparse_d_create_csr

C_FLOAT_COMPLEX for mkl_sparse_c_create_csr

C_DOUBLE_COMPLEX for mkl_sparse_z_create_csr

Array containing non-zero elements of the matrix *A*. Its length is equal to length of the *col_indx* array.

Refer to *values* array description in CSR Format for more details.

Output Parameters

A

SPARSE_MATRIX_T.

Handle containing internal data for subsequent Inspector-executor Sparse BLAS operations.

stat

INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_create_csc

Creates a handle for a CSC format matrix.

Syntax

stat = mkl_sparse_s_create_csc (*A*, *indexing*, *rows*, *cols*, *cols_start*, *cols_end*, *row_indx*, *values*)

stat = mkl_sparse_d_create_csc (*A*, *indexing*, *rows*, *cols*, *cols_start*, *cols_end*, *row_indx*, *values*)stat = mkl_sparse_c_create_csc (*A*, *indexing*, *rows*, *cols*, *cols_start*, *cols_end*, *row_indx*, *values*)stat = mkl_sparse_z_create_csc (*A*, *indexing*, *rows*, *cols*, *cols_start*, *cols_end*, *row_indx*, *values*)

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_?_create_csc` routine creates a handle for an *m*-by-*k* matrix *A* in CSC format.

> **NOTE**
> The input arrays provided are left unchanged except for the call to mkl_sparse_order, which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use mkl_sparse_copy.

Input Parameters

indexing

`sparse_index_base_t`.

Indicates how input arrays are indexed.

`SPARSE_INDEX_BASE_ZERO`	Zero-based (C-style) indexing: indices start at 0.
`SPARSE_INDEX_BASE_ONE`	One-based (Fortran-style) indexing: indices start at 1.

rows

`C_INT`.

Number of rows of the matrix *A*.

cols

`C_INT`.

Number of columns of the matrix *A*.

cols_start

`C_INT`.

Array of length at least *m*. This array contains col indices, such that *cols_start*(*i*) - *ind* is the first index of col *i* in the arrays *values* and *row_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerB* array description in CSC Format for more details.

cols_end

`C_INT`.

Array of at least length *m*. This array contains col indices, such that *cols_end*(*i*) - *ind* – 1 is the last index of col *i* in the arrays *values* and *row_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerE* array description in CSC Format for more details.

row_indx

`C_INT`.

For one-based indexing, array containing the row indices plus one for each non-zero element of the matrix *A*. For zero-based indexing, array containing the row indices for each non-zero element of the matrix *A*. Its length is at least *cols_end*(*cols* – 1) - *ind*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

values

`C_FLOAT` for `mkl_sparse_s_create_csc`

`C_DOUBLE` for `mkl_sparse_d_create_csc`

`C_FLOAT_COMPLEX` for `mkl_sparse_c_create_csc`

C_DOUBLE_COMPLEX for mkl_sparse_z_create_csc

Array containing non-zero elements of the matrix *A*. Its length is equal to length of the *row_indx* array.

Refer to *values* array description in CSC Format for more details.

Output Parameters

A SPARSE_MATRIX_T.

Handle containing internal data.

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_create_coo

Creates a handle for a matrix in COO format.

Syntax

```
stat = mkl_sparse_s_create_coo (A, indexing, rows, cols, nnz, row_indx, col_indx, values)
stat = mkl_sparse_d_create_coo (A, indexing, rows, cols, nnz, row_indx, col_indx, values)
stat = mkl_sparse_c_create_coo (A, indexing, rows, cols, nnz, row_indx, col_indx, values)
stat = mkl_sparse_z_create_coo (A, indexing, rows, cols, nnz, row_indx, col_indx, values)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_create_coo routine creates a handle for an *m*-by-*k* matrix *A* in COO format.

NOTE
The input arrays provided are left unchanged except for the call to mkl_sparse_order, which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use mkl_sparse_copy.

Input Parameters

indexing

sparse_index_base_t.

Indicates how input arrays are indexed.

SPARSE_INDEX_BASE_ZERO	Zero-based (C-style) indexing: indices start at 0.
SPARSE_INDEX_BASE_ONE	One-based (Fortran-style) indexing: indices start at 1.

rows

C_INT.

Number of rows of matrix *A*.

cols

C_INT.

Number of columns of matrix *A*.

nnz

C_INT.

Specifies the number of non-zero elements of the matrix *A*.

Refer to *nnz* description in Coordinate Format for more details.

row_indx

C_INT.

Array of length *nnz*, containing the row indices for each non-zero element of matrix *A*.

Refer to *rows* array description in Coordinate Format for more details.

col_indx

C_INT.

Array of length *nnz*, containing the column indices for each non-zero element of matrix *A*.

Refer to *columns* array description in Coordinate Format for more details.

values

C_FLOAT for mkl_sparse_s_create_coo

C_DOUBLE for mkl_sparse_d_create_coo

C_FLOAT_COMPLEX for mkl_sparse_c_create_coo

C_DOUBLE_COMPLEX for mkl_sparse_z_create_coo

Array of length *nnz*, containing the non-zero elements of matrix *A* in arbitrary order.

Refer to *values* array description in Coordinate Format for more details.

Output Parameters

A

SPARSE_MATRIX_T.

Handle containing internal data.

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_create_bsr

Creates a handle for a matrix in BSR format.

Syntax

```
stat = mkl_sparse_s_create_bsr (A, indexing, block_layout, rows, cols, block_size, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_d_create_bsr (A, indexing, block_layout, rows, cols, block_size, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_c_create_bsr (A, indexing, block_layout, rows, cols, block_size, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_z_create_bsr (A, indexing, block_layout, rows, cols, block_size, rows_start, rows_end, col_indx, values)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_create_bsr routine creates a handle for an *m*-by-*k* matrix *A* in BSR format.

NOTE
The input arrays provided are left unchanged except for the call to mkl_sparse_order, which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use mkl_sparse_copy.

Input Parameters

indexing sparse_index_base_t.

Indicates how input arrays are indexed.

SPARSE_INDEX_BASE_ZERO	Zero-based (C-style) indexing: indices start at 0.
SPARSE_INDEX_BASE_ONE	One-based (Fortran-style) indexing: indices start at 1.

block_layout

sparse_index_base_t.

Specifies layout of blocks:

SPARSE_LAYOUT_ROW_MAJOR	Storage of elements of blocks uses row major layout.
SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements of blocks uses column major layout.

rows

C_INT.

Number of block rows of matrix *A*.

cols

C_INT.

Number of block columns of matrix *A*.

block_size

C_INT.

Size of blocks in matrix *A*.

rows_start

C_INT.

Array of length *m*. This array contains row indices, such that *rows_start*(*i*) - *ind* is the first index of block row *i* in the arrays *values* and *col_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerB* array description in CSR Format for more details.

rows_end

C_INT.

Array of length *m*. This array contains row indices, such that *rows_end*(*i*) - *ind*– 1 is the last index of block row *i* in the arrays *values* and *col_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerE* array description in CSR Format for more details.

col_indx

C_INT.

For one-based indexing, array containing the column indices plus one for each non-zero block of the matrix *A*. For zero-based indexing, array containing the column indices for each non-zero block of the matrix *A*. Its length is *rows_end*(*rows* – 1) - *ind*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

values

C_FLOAT for mkl_sparse_s_create_bsr

C_DOUBLE for mkl_sparse_d_create_bsr

C_FLOAT_COMPLEX for mkl_sparse_c_create_bsr

C_DOUBLE_COMPLEX for mkl_sparse_z_create_bsr

Array containing non-zero elements of the matrix *A*. Its length is equal to length of the *col_indx* array multiplied by *block_size***block_size*.

Refer to the *values* array description in BSR Format for more details.

Output Parameters

A — SPARSE_MATRIX_T.

Handle containing internal data.

stat — INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_copy

Creates a copy of a matrix handle.

Syntax

```
stat = mkl_sparse_copy (source, descr, dest)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_copy routine creates a copy of a matrix handle.

> **NOTE**
> Currently, the mkl_sparse_copy routine does not support the descriptor argument and creates an exact (deep) copy of the input matrix.

Input Parameters

source — SPARSE_MATRIX_T.

Specifies handle containing internal data.

descr

`MATRIX_DESCR`.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

`SPARSE_MATRIX_TYPE_GENERAL`	The matrix is processed as is.
`SPARSE_MATRIX_TYPE_SYMMETRIC`	The matrix is symmetric (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_HERMITIAN`	The matrix is Hermitian (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_TRIANGULAR`	The matrix is triangular (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_DIAGONAL`	The matrix is diagonal (only diagonal elements are processed).
`SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR`	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
`SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL`	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

`SPARSE_FILL_MODE_LOWER`	The lower triangular matrix part is processed.
`SPARSE_FILL_MODE_UPPER`	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

`SPARSE_DIAG_NON_UNIT`	Diagonal elements might not be equal to one.
`SPARSE_DIAG_UNIT`	Diagonal elements are equal to one.

Output Parameters

dest

`SPARSE_MATRIX_T` for `mkl_sparse_copy`

Handle containing internal data.

stat

`INTEGER`

Value indicating whether the operation was successful or not, and why:

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_destroy

Frees memory allocated for matrix handle.

Syntax

```
stat = mkl_sparse_destroy (A)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_destroy routine frees memory allocated for matrix handle.

NOTE
You must free memory allocated for matrices after completing use of them. The mkl_sparse_destroy routine provides a utility to do so.

Input Parameters

A SPARSE_MATRIX_T.

Handle containing internal data.

Output Parameters

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_convert_csr

Converts internal matrix representation to CSR format.

Syntax

```
stat = mkl_sparse_convert_csr (source, operation, dest)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_convert_csr routine converts internal matrix representation to CSR format.

When the source matrix is in COO format, the routine performs a sum reduction on duplicate elements.

Input Parameters

source SPARSE_MATRIX_T.

Handle containing internal data.

operation C_INT.

Specifies operation op() on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.
SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.

Output Parameters

dest SPARSE_MATRIX_T.

Handle containing internal data.

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_convert_bsr

Converts internal matrix representation to BSR format or changes BSR block size.

Syntax

```
stat = mkl_sparse_convert_bsr (source, block_size, block_layout, operation, dest)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_convert_bsr routine converts internal matrix representation to BSR format or changes BSR block size.

When the source matrix is in COO format, the routine performs a sum reduction on duplicate elements.

Input Parameters

source

SPARSE_MATRIX_T.

Handle containing internal data.

block_size

C_INT.

Size of the block in the output structure.

block_layout

sparse_index_base_t.

Specifies layout of blocks:

SPARSE_LAYOUT_ROW_MAJOR	Storage of elements of blocks uses row major layout.
SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements of blocks uses column major layout.

operation

C_INT.

Specifies operation op() on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.

SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.

Output Parameters

dest

SPARSE_MATRIX_T.

Handle containing internal data.

stat

INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_export_csr

Exports CSR matrix from internal representation.

Syntax

```
stat = mkl_sparse_s_export_csr (source, indexing, rows, cols, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_d_export_csr (source, indexing, rows, cols, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_c_export_csr (source, indexing, rows, cols, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_z_export_csr (source, indexing, rows, cols, rows_start, rows_end, col_indx, values)
```

Include Files

- mkl_spblas.f90

Description

If the matrix specified by the *source* handle is in CSR format, the `mkl_sparse_?_export_csr` routine exports an *m*-by-*k* matrix *A* in CSR format matrix from the internal representation. The routine returns pointers to the internal representation and does not allocate additional memory.

If the matrix is not already in CSR format, the routine returns `SPARSE_STATUS_INVALID_VALUE`.

Input Parameters

source `SPARSE_MATRIX_T`.

Handle containing internal data.

Output Parameters

indexing `sparse_index_base_t`.

Indicates how input arrays are indexed.

`SPARSE_INDEX_BASE_ZERO`	Zero-based (C-style) indexing: indices start at 0.
`SPARSE_INDEX_BASE_ONE`	One-based (Fortran-style) indexing: indices start at 1.

rows `C_INT`.

Number of rows of the matrix *source*.

cols `C_INT`.

Number of columns of the matrix *source*.

rows_start `C_INT`.

Pointer to array of length *m*. This array contains row indices, such that *rows_start*(*i*) - *ind* is the first index of row *i* in the arrays *values* and *col_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerB* array description in CSR Format for more details.

rows_end `C_INT`.

Pointer to array of length *m*. This array contains row indices, such that *rows_end*(*i*) - *ind* - 1 is the last index of row *i* in the arrays *values* and *col_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerE* array description in CSR Format for more details.

col_indx `C_INT`.

For one-based indexing, pointer to array containing the column indices plus one for each non-zero element of the matrix *source*. For zero-based indexing, pointer to array containing the column indices for each non-zero element of the matrix *source*. Its length is *rows_end*(*rows* – 1) - *ind*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

values `C_FLOAT` for `mkl_sparse_s_export_csr`

C_DOUBLE for mkl_sparse_d_export_csr

C_FLOAT_COMPLEX for mkl_sparse_c_export_csr

C_DOUBLE_COMPLEX for mkl_sparse_z_export_csr

Pointer to array containing non-zero elements of the matrix *A*. Its length is equal to length of the *col_indx* array.

Refer to *values* array description in CSR Format for more details.

Output Parameters

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_export_csc

Exports CSC matrix from internal representation.

Syntax

```
stat = mkl_sparse_s_export_csc (source, indexing, rows, cols, cols_start, cols_end, row_indx, values)
stat = mkl_sparse_d_export_csc (source, indexing, rows, cols, cols_start, cols_end, row_indx, values)
stat = mkl_sparse_c_export_csc (source, indexing, rows, cols, cols_start, cols_end, row_indx, values)
stat = mkl_sparse_z_export_csc (source, indexing, rows, cols, cols_start, cols_end, row_indx, values)
```

Include Files

- mkl_spblas.f90

Description

If the matrix specified by the *source* handle is in CSC format, the mkl_sparse_?_export_csc routine exports an *m*-by-*k* matrix *A* in CSC format matrix from the internal representation. The routine returns pointers to the internal representation and does not allocate additional memory.

If the matrix is not already in CSC format, the routine returns SPARSE_STATUS_INVALID_VALUE.

Input Parameters

source SPARSE_MATRIX_T.

Handle containing internal data.

Output Parameters

indexing sparse_index_base_t.

Indicates how input arrays are indexed.

SPARSE_INDEX_BASE_ZERO	Zero-based (C-style) indexing: indices start at 0.
SPARSE_INDEX_BASE_ONE	One-based (Fortran-style) indexing: indices start at 1.

rows C_INT.

Number of rows of the matrix *source*.

cols C_INT.

Number of columns of the matrix *source*.

cols_start C_INT.

Array of length *m*. This array contains column indices, such that *cols_start*(*i*) - *cols_start*(1) is the first index of column *i* in the arrays *values* and *row_indx*.

Refer to *pointerb* array description in csc Format for more details.

cols_end C_INT.

Pointer to array of length *m*. This array contains row indices, such that *cols_end*(*i*) - *cols_start*(1) - 1 is the last index of column *i* in the arrays *values* and *row_indx*.

Refer to *pointerE* array description in csc Format for more details.

row_indx C_INT.

For one-based indexing, pointer to array containing the row indices plus one for each non-zero element of the matrix *source*. For zero-based indexing, pointer to array containing the row indices for each non-zero element of the matrix *source*. Its length is *cols_end*(*cols* – 1) - *cols_start*(1).

values C_FLOAT for mkl_sparse_s_export_csc

C_DOUBLE for mkl_sparse_d_export_csc

C_FLOAT_COMPLEX for mkl_sparse_c_export_csc

C_DOUBLE_COMPLEX for mkl_sparse_z_export_csc

Pointer to array containing non-zero elements of the matrix *A*. Its length is equal to length of the *row_indx* array.

Refer to *values* array description in csc Format for more details.

Output Parameters

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_export_bsr

Exports BSR matrix from internal representation.

Syntax

```
stat = mkl_sparse_s_export_bsr (source, indexing, block_layout, rows, cols, block_size, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_d_export_bsr (source, indexing, block_layout, rows, cols, block_size, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_c_export_bsr (source, indexing, block_layout, rows, cols, block_size, rows_start, rows_end, col_indx, values)
stat = mkl_sparse_z_export_bsr (source, indexing, block_layout, rows, cols, block_size, rows_start, rows_end, col_indx, values)
```

Include Files

- mkl_spblas.f90

Description

If the matrix specified by the *source* handle is in BSR format, the mkl_sparse_?_export_bsr routine exports an (*block_size* * *rows*)-by-(*block_size* * *cols*) matrix *A* in BSR format from the internal representation. The routine returns pointers to the internal representation and does not allocate additional memory.

If the matrix is not already in BSR format, the routine returns SPARSE_STATUS_INVALID_VALUE.

Input Parameters

source SPARSE_MATRIX_T.

Handle containing internal data.

Output Parameters

indexing

`sparse_index_base_t`.

Indicates how input arrays are indexed.

`SPARSE_INDEX_BASE_ZERO`	Zero-based (C-style) indexing: indices start at 0.
`SPARSE_INDEX_BASE_ONE`	One-based (Fortran-style) indexing: indices start at 1.

block_layout

`sparse_index_base_t`.

Specifies layout of blocks:

`SPARSE_LAYOUT_ROW_MAJOR`	Storage of elements of blocks uses row major layout.
`SPARSE_LAYOUT_COLUMN_MAJOR`	Storage of elements of blocks uses column major layout.

rows

`C_INT`.

Number of block rows of the matrix *source*.

cols

`C_INT`.

Number of block columns of matrix *source*.

block_size

`C_INT`.

Size of the square block in matrix *source*.

rows_start

`C_INT`.

Pointer to array of length *rows*. This array contains row indices, such that *rows_start*(*i*) - *ind* is the first index of block row *i* in the arrays *values* and *col_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerB* array description in BSR Format for more details.

rows_end

`C_INT`.

Pointer to array of length *rows*. This array contains row indices, such that *rows_end*(*i*) - *ind* - 1 is the last index of block row *i* in the arrays *values* and *col_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerE* array description in BSR Format for more details.

col_indx

`C_INT`.

For one-based indexing, pointer to array containing the column indices plus one for each non-zero blocks of the matrix *source*. For zero-based indexing, pointer to array containing the column indices for each non-zero blocks of the matrix *source*. Its length is *rows_end*(*rows* – 1) - *ind*(1). *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

values

`C_FLOAT` for `mkl_sparse_s_export_bsr`

`C_DOUBLE` for `mkl_sparse_d_export_bsr`

C_FLOAT_COMPLEX for mkl_sparse_c_export_bsr

C_DOUBLE_COMPLEX for mkl_sparse_z_export_bsr

Pointer to array containing non-zero elements of matrix *source*. Its length is equal to length of the *col_indx* array multiplied by *block_size***block_size*.

Refer to the *values* array description in BSR Format for more details.

Output Parameters

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_set_value

Changes a single value of matrix in internal representation.

Syntax

```
stat = mkl_sparse_s_set_value (A , row, col, value);
stat = mkl_sparse_d_set_value (A, row, col, value );
stat = mkl_sparse_c_set_value (A , row, col, value);
stat = mkl_sparse_z_set_value (A , row, col, value);
```

Include Files

- mkl_spblas.f90

Description

Use the mkl_sparse_?_set_value routine to change a single value of a matrix in the internal Inspector-executor Sparse BLAS format. The value should already be presented in a matrix structure.

Input Parameters

A SPARSE_MATRIX_T.

Specifies handle containing internal data.

row C_INT .

Indicates row of matrix in which to set value.

col C_INT .

Indicates column of matrix in which to set value.

value C_FLOAT for mkl_sparse_s_create_csr

C_DOUBLE for mkl_sparse_d_create_csr

C_FLOAT_COMPLEX for mkl_sparse_c_create_csr

C_DOUBLE_COMPLEX for mkl_sparse_z_create_csr

Indicates value

Output Parameters

A Handle containing modified internal data.

stat INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.

mkl_sparse_?_update_values

Changes all or selected matrix values in internal representation.

Syntax

NOTE
This routine is supported for sparse matrices in BSR format only.

```
status = mkl_sparse_s_update_values (A, values, indx, indy, values)
status = mkl_sparse_d_update_values (A, values, indx, indy, values)
status = mkl_sparse_c_update_values (A, values, indx, indy, values)
status = mkl_sparse_z_update_values (A, values, indx, indy, values)
```

Include Files

- mkl_spblas.f90

Description

Use the `mkl_sparse_?_update_values` routine to change all or selected values of a matrix in the internal Inspector-Executor Sparse BLAS format.

The values to be updated should already be present in the matrix structure.

- To change selected values, you must provide an array values (with new values) and also the corresponding row and column indices for each value via *indx* and *indy* arrays as well as the overall number of changed elements *nvalues*.

  So that, for example, to change A(0, 0) to 1 and A(0, 1) to 2, pass the following input parameters: *nvalues* = 2, *indx* = {0, 0}, *indy* = {0, 1} and *values* = {1, 2}.
- To change all the values in the matrix, provide the *values* array and explicitly set *nvalues* to 0 or the actual number of non zero elements. There is no need to supply *indx* and *indy* arrays.

Input Parameters

A

`SPARSE_MATRIX_T`.

Specifies handle containing internal data.

nvalues

`C_INT` .

Total number of elements changed.

indx

`C_INT` .

Row indices for the new values.

> **NOTE**
> Currently, only updating the full matrix is supported. Set *indx* and *indy* as NULL.

indy

`C_INT` .

Column indices for the new values.

> **NOTE**
> Currently, only updating the full matrix is supported. Set *indx* and *indy* as NULL.

values

`C_FLOAT` for `mkl_sparse_s_update_values`

`C_DOUBLE` for `mkl_sparse_d_update_values`

`C_FLOAT_COMPLEX` for `mkl_sparse_c_update_values`

`C_DOUBLE_COMPLEX` for `mkl_sparse_z_update_values`

New values.

Output Parameters

A

`SPARSE_MATRIX_T`.

Handle containing modified internal data.

status

`INTEGER`

Value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_order

Performs ordering of column indexes of the matrix in CSR format

Syntax

```
stat = mkl_sparse_order(csrA)
```

Include Files

- mkl_spblas.f90

Description

Use the mkl_sparse_order routine to perform ordering of column indexes of the matrix in CSR format.

Input Parameters

csrA	SPARSE_MATRIX_T. CSR data

Output Parameters

csrA	Handle containing modified internal data.
stat	INTEGER. Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.

SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.

Inspector-Executor Sparse BLAS Analysis Routines

Analysis Routines and Their Data Types

Routine or Function Group	Description
mkl_sparse_set_lu_smoother_hint	Provides and estimate of the number and type of upcoming calls to LU smoother functionality.
mkl_sparse_set_mv_hint	Provides estimate of number and type of upcoming matrix-vector operations.
mkl_sparse_set_sv_hint	Provides estimate of number and type of upcoming triangular system solver operations.
mkl_sparse_set_mm_hint	Provides estimate of number and type of upcoming matrix-matrix multiplication operations.
mkl_sparse_set_sm_hint	Provides estimate of number and type of upcoming triangular matrix solve with multiple right hand sides operations.
mkl_sparse_set_dotmv_hint	Sets estimate of the number and type of upcoming matrix-vector operations.
mkl_sparse_set_symgs_hint	Sets estimate of number and type of upcoming `mkl_sparse_?_symgs` operations.
mkl_sparse_set_sorv_hint	Sets estimate of number and type of upcoming `mkl_sparse_?_symgs` operations.
mkl_sparse_set_memory_hint	Provides memory requirements for performance optimization purposes.
mkl_sparse_optimize	Analyzes matrix structure and performs optimizations using the hints provided in the handle.

mkl_sparse_set_lu_smoother_hint

Provides an estimate of the number and type of upcoming calls to LU smoother functionality.

Syntax

```
status = mkl_sparse_set_lu_smoother_hint (A, operation, descr, expected_calls)
```

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_set_lu_smoother_hint` function provides subsequent Inspector-Executor Sparse BLAS calls an estimate of the number of upcoming calls to the `lu_smoother` routine that ultimately may influence the optimizations applied and specifies whether or not to perform an operation on the matrix.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

operation

`C_INT` .

Specifies the operation *op()* on input matrix.

`SPARSE_OPERATION_NON_TRANSPOSE`	Non-transpose, `op(A)` = A.
`SPARSE_OPERATION_TRANSPOSE`	Transpose, `op(A)` = A^T.
`SPARSE_OPERATION_CONJUGATE_TRANSPOSE`	Conjugate transpose, `op(A)` = A^H.

descr

`MATRIX_DESCR` .

Structure specifying sparse matrix properties.

`sparse_matrix_type_t`*type* - Specifies the type of a sparse matrix:

`SPARSE_MATRIX_TYPE_GENERAL`	The matrix is processed as is.
`SPARSE_MATRIX_TYPE_SYMMETRIC`	The matrix is symmetric (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_HERMITIAN`	The matrix is Hermitian (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_TRIANGULAR`	The matrix is triangular (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_DIAGONAL`	The matrix is diagonal (only diagonal elements are processed).
`SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR`	The matrix is block-triangular (only the requested triangle is processed). Applies to BSR format only.
`SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL`	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`sparse_fill_mode_t`*mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

	`SPARSE_FILL_MODE_LOWER` The lower triangular matrix part is processed. `SPARSE_FILL_MODE_UPPER` The upper triangular matrix part is processed. `sparse_diag_type_t`*diag* - Specifies the diagonal type for non-general matrices: `SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one. `SPARSE_DIAG_UNIT` Diagonal elements are equal to one.
expected_calls	`C_INT` . Number of expected calls to execution routine.
A	`SPARSE_MATRIX_T`. Handle containing internal data.
status	`INTEGER` Value indicating whether the operation was successful or not, and why. `SPARSE_STATUS_SUCCESS` The operation was successful. `SPARSE_STATUS_NOT_INITIALIZED` The routine encountered an empty handle or matrix array. `SPARSE_STATUS_ALLOC_FAILED` Internal memory allocation failed. `SPARSE_STATUS_INVALID_VALUE` The input parameters contain an invalid value. `SPARSE_STATUS_EXECUTION_FAILED` Execution failed. `SPARSE_STATUS_INTERNAL_ERROR` An error in algorithm implementation occurred. `SPARSE_STATUS_NOT_SUPPORTED` The requested operation is not supported.

mkl_sparse_set_mv_hint

Provides estimate of number and type of upcoming matrix-vector operations.

Syntax

```
stat = mkl_sparse_set_mv_hint (A, operation, descr, expected_calls)
```

Include Files

- `mkl_spblas.f90`

Description

Use the `mkl_sparse_set_mv_hint` routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming matrix-vector multiplication operations for performance optimization, and specify whether or not to perform an operation on the matrix.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

operation

C_INT.

Specifies operation op() on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.
SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.

descr

MATRIX_DESCR.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

	SPARSE_DIAG_NON_UNIT	Diagonal elements might not be equal to one.
	SPARSE_DIAG_UNIT	Diagonal elements are equal to one.
expected_calls	C_INT.	
	Number of expected calls to execution routine.	

Output Parameters

A	SPARSE_MATRIX_T.	
	Handle containing internal data.	
stat	INTEGER	
	Value indicating whether the operation was successful or not, and why:	
	SPARSE_STATUS_SUCCESS	The operation was successful.
	SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
	SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
	SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
	SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
	SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
	SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_set_sv_hint

Provides estimate of number and type of upcoming triangular system solver operations.

Syntax

```
stat = mkl_sparse_set_sv_hint (A, operation, descr, expected_calls)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_set_sv_hint routine provides an estimate of the number of upcoming triangular system solver operations and type of these operations for performance optimization.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Product and Performance Information
Notice revision #20201201

Input Parameters

operation C_INT.

Specifies operation op() on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.
SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.

descr MATRIX_DESCR.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT	Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT	Diagonal elements are equal to one.

expected_calls C_INT.

Number of expected calls to execution routine.

Output Parameters

A SPARSE_MATRIX_T.

Handle containing internal data.

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_set_mm_hint

Provides estimate of number and type of upcoming matrix-matrix multiplication operations.

Syntax

```
stat = mkl_sparse_set_mm_hint (A, operation, descr, layout, dense_matrix_size, expected_calls)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_set_mm_hint routine provides an estimate of the number of upcoming matrix-matrix multiplication operations and type of these operations for performance optimization purposes.

Input Parameters

operation

C_INT.

Specifies operation op() on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.
SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.

descr

MATRIX_DESCR.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT	Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT	Diagonal elements are equal to one.

layout

C_INT.

Specifies layout of elements:

	`SPARSE_LAYOUT_COLUMN_MAJOR` Storage of elements uses column major layout.
	`SPARSE_LAYOUT_ROW_MAJOR` Storage of elements uses row major layout.
dense_matrix_size	`C_INT`. Number of columns in dense matrix.
expected_calls	`C_INT`. Number of expected calls to execution routine.

Output Parameters

A	`SPARSE_MATRIX_T`. Handle containing internal data.
stat	`INTEGER` Value indicating whether the operation was successful or not, and why:

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.
`SPARSE_STATUS_EXECUTION_FAILED`	Execution failed.
`SPARSE_STATUS_INTERNAL_ERROR`	An error in algorithm implementation occurred.
`SPARSE_STATUS_NOT_SUPPORTED`	The requested operation is not supported.

mkl_sparse_set_sm_hint

Provides estimate of number and type of upcoming triangular matrix solve with multiple right hand sides operations.

Syntax

`stat = mkl_sparse_set_sm_hint (A, operation, descr, layout, dense_matrix_size, expected_calls)`

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_set_sm_hint` routine provides an estimate of the number of upcoming triangular matrix solve with multiple right hand sides operations and type of these operations for performance optimization purposes.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

operation

`C_INT`.

Specifies operation `op()` on input matrix.

`SPARSE_OPERATION_NON_TRANSPOSE`	Non-transpose, `op(`A`) = `A.
`SPARSE_OPERATION_TRANSPOSE`	Transpose, `op(`A`) = `A^T.
`SPARSE_OPERATION_CONJUGATE_TRANSPOSE`	Conjugate transpose, `op(`A`) = `A^H.

descr

`MATRIX_DESCR`.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

`SPARSE_MATRIX_TYPE_GENERAL`	The matrix is processed as is.
`SPARSE_MATRIX_TYPE_SYMMETRIC`	The matrix is symmetric (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_HERMITIAN`	The matrix is Hermitian (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_TRIANGULAR`	The matrix is triangular (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_DIAGONAL`	The matrix is diagonal (only diagonal elements are processed).
`SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR`	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
`SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL`	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

	SPARSE_FILL_MODE_LOWER — The lower triangular matrix part is processed.
	SPARSE_FILL_MODE_UPPER — The upper triangular matrix part is processed.
	diag - Specifies diagonal type for non-general matrices:
	SPARSE_DIAG_NON_UNIT — Diagonal elements might not be equal to one.
	SPARSE_DIAG_UNIT — Diagonal elements are equal to one.
layout	C_INT.
	Specifies layout of elements:
	SPARSE_LAYOUT_COLUMN_MAJOR — Storage of elements uses column major layout.
	SPARSE_LAYOUT_ROW_MAJOR — Storage of elements uses row major layout.
dense_matrix_size	C_INT.
	Number of right-hand-side.
expected_calls	C_INT.
	Number of expected calls to execution routine.

Output Parameters

A	SPARSE_MATRIX_T.
	Handle containing internal data.
stat	INTEGER
	Value indicating whether the operation was successful or not, and why:
	SPARSE_STATUS_SUCCESS — The operation was successful.
	SPARSE_STATUS_NOT_INITIALIZED — The routine encountered an empty handle or matrix array.
	SPARSE_STATUS_ALLOC_FAILED — Internal memory allocation failed.
	SPARSE_STATUS_INVALID_VALUE — The input parameters contain an invalid value.
	SPARSE_STATUS_EXECUTION_FAILED — Execution failed.
	SPARSE_STATUS_INTERNAL_ERROR — An error in algorithm implementation occurred.
	SPARSE_STATUS_NOT_SUPPORTED — The requested operation is not supported.

mkl_sparse_set_dotmv_hint

Sets estimate of the number and type of upcoming matrix-vector operations.

Syntax

```
stat = mkl_sparse_set_dotmv_hint (A, operation, descr, layout, expected_calls)
```

Include Files

- mkl_spblas.f90

Description

Use the mkl_sparse_set_dotmv_hint routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming matrix-vector multiplication operations for performance optimization, and specify whether or not to perform an operation on the matrix.

Input Parameters

operation

C_INT.

Specifies the operation performed on matrix *A*.

If *operation* = SPARSE_OPERATION_NON_TRANSPOSE, op(*A*) = *A*.

If *operation* = SPARSE_OPERATION_TRANSPOSE, op(*A*) = A^T.

If *operation* = SPARSE_OPERATION_CONJUGATE_TRANSPOSE, op(*A*) = A^H.

descr

MATRIX_DESCR.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

	`SPARSE_FILL_MODE_LOWER` The lower triangular matrix part is processed. `SPARSE_FILL_MODE_UPPER` The upper triangular matrix part is processed. *diag* - Specifies diagonal type for non-general matrices: `SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one. `SPARSE_DIAG_UNIT` Diagonal elements are equal to one.
expected_calls	`C_INT`. Expected number of calls to the execution routine.

Output Parameters

A	`SPARSE_MATRIX_T`. Handle containing internal data.
stat	`INTEGER` Value indicating whether the operation was successful or not, and why: `SPARSE_STATUS_SUCCESS` The operation was successful. `SPARSE_STATUS_NOT_INITIALIZED` The routine encountered an empty handle or matrix array. `SPARSE_STATUS_ALLOC_FAILED` Internal memory allocation failed. `SPARSE_STATUS_INVALID_VALUE` The input parameters contain an invalid value. `SPARSE_STATUS_EXECUTION_FAILED` Execution failed. `SPARSE_STATUS_INTERNAL_ERROR` An error in algorithm implementation occurred. `SPARSE_STATUS_NOT_SUPPORTED` The requested operation is not supported.

mkl_sparse_set_symgs_hint

Syntax

Sets estimate of number and type of upcoming `mkl_sparse_?_symgs` operations.

`stat =mkl_sparse_set_symgs_hint (A, operation, descr, layout, dense_matrix_size, expected_calls)`

Include Files

- `mkl_spblas.f90`

Description

Use the `mkl_sparse_set_symgs_hint` routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming symmetric Gauss-Zeidel preconditioner operations for performance optimization, and specify whether or not to perform an operation on the matrix.

Input Parameters

operation

`C_INT`.

Specifies the operation performed on matrix *A*.

If *operation* = `SPARSE_OPERATION_NON_TRANSPOSE`, `op(`*A*`) = `*A*.

If *operation* = `SPARSE_OPERATION_TRANSPOSE`, `op(`*A*`) = `A^T.

If *operation* = `SPARSE_OPERATION_CONJUGATE_TRANSPOSE`, `op(`*A*`) = `A^H.

descr

`MATRIX_DESCR`.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

`SPARSE_MATRIX_TYPE_GENERAL`	The matrix is processed as is.
`SPARSE_MATRIX_TYPE_SYMMETRIC`	The matrix is symmetric (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_HERMITIAN`	The matrix is Hermitian (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_TRIANGULAR`	The matrix is triangular (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_DIAGONAL`	The matrix is diagonal (only diagonal elements are processed).
`SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR`	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
`SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL`	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

`SPARSE_FILL_MODE_LOWER`	The lower triangular matrix part is processed.
`SPARSE_FILL_MODE_UPPER`	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

`SPARSE_DIAG_NON_UNIT`	Diagonal elements might not be equal to one.
`SPARSE_DIAG_UNIT`	Diagonal elements are equal to one.

diag

`C_INT`.

Specifies diagonal type for non-general matrices

mode C_INT.

Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices.

type C_INT.

Specifies the type of a sparse matrix.

expected_calls C_INT.

Estimate of the number to the execution routine.

Output Parameters

A SPARSE_MATRIX_T.

Handle containing internal data.

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_set_sorv_hint

Sets an estimate of the number and type of upcoming mkl_sparse_?_sorv *operations.*

Syntax

```
stat = sparse_status_t mkl_sparse_set_sorv_hint(type, A, descr, expected_calls)
```

Include Files

- mkl_spblas.f90

Description

Use the `mkl_sparse_set_sorv_hint` routine to provide the Inspector-Executor Sparse BLAS API an estimate of the number of upcoming forward/backward sweeps or symmetric SOR preconditioner operations for performance optimization.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

type `SPARSE_MATRIX_T`.

Specifies the operation performed by the SORV preconditioner.

`SPARSE_SOR_FORWARD` Performs forward sweep as defined by:

$$(\omega*L+D)*x\text{^}1 = (D-\omega*D-\omega*U)*x\text{^}0 + \omega*b$$

`SPARSE_SOR_BACKWARD` Performs backward sweep as defined by:

$$(\omega*U+D)*x\text{^}1 = (D-\omega*D-\omega*L)*x\text{^}0 + \omega*b$$

`SPARSE_SOR_SYMMETRIC` Preconditioner matrix could be expressed as:

$$\frac{\omega}{2-\omega}\left(\frac{1}{\omega}D+L\right)D^{-1}\left(\frac{1}{\omega}D+L\right)^{T}$$

descr `MATRIX_DESCR`.

Structure specifying sparse matrix properties.

`SPARSE_MATRIX_T type` Specifies the type of a sparse matrix:

- `SPARSE_MATRIX_TYPE_GENERAL`

  The matrix is processed as-is.
- `SPARSE_MATRIX_TYPE_SYMMETRIC`

  The matrix is symmetric (only the requested triangle is processed).
- `SPARSE_MATRIX_TYPE_HERMITIAN`

  The matrix is Hermitian (only the requested triangle is processed).
- `SPARSE_MATRIX_TYPE_TRIANGULAR`

  The matrix is triangular (only the requested triangle is processed).
- `SPARSE_MATRIX_TYPE_DIAGONAL`

  The matrix is diagonal (only diagonal elements are processed).
- `SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR`

The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.

- `SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL`

  The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`C_INT mode` — Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

- `SPARSE_FILL_MODE_LOWER`

  The lower triangular matrix part is processed.
- `SPARSE_FILL_MODE_UPPER`

  The upper triangular matrix part is processed.

`SPARSE_MATRIX_TYPE_DIAGONAL diag` — Specifies diagonal type for non-general matrices:

- `SPARSE_DIAG_NON_UNIT`

  Diagonal elements might not be equal to one.
- `SPARSE_DIAG_UNIT`

  Diagonal elements are equal to one.

A — `SPARSE_MATRIX_T`.

Handle containing internal data.

expected_calls — `INTEGER`.

Estimate of the number of calls to the execution routine.

Output Parameters

A — `SPARSE_MATRIX_T`.

Handle containing internal data.

stat — `INTEGER`

Value indicating whether the operation was successful or not, and why:

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_set_memory_hint

Provides memory requirements for performance optimization purposes.

Syntax

`stat = mkl_sparse_set_memory_hint (A, policy)`

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_set_memory_hint` routine allocates additional memory for further performance optimization purposes.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Input Parameters

policy — C_INT.

Specify memory utilization policy for optimization routine using these types:

SPARSE_MEMORY_NONE	Routine can allocate memory only for auxiliary structures (such as for workload balancing); the amount of memory is proportional to vector size.
SPARSE_MEMORY_AGGRESSIVE	Default. Routine can allocate memory up to the size of matrix *A* for converting into the appropriate sparse format.

Output Parameters

A — SPARSE_MATRIX_T.

Handle containing internal data.

stat — INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_optimize

Analyzes matrix structure and performs optimizations using the hints provided in the handle.

Syntax

```
stat = mkl_sparse_optimize (A)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_optimize routine analyzes matrix structure and performs optimizations using the hints provided in the handle. Generally, specifying a higher number of expected operations allows for more aggressive and time consuming optimizations.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Input Parameters

A SPARSE_MATRIX_T.

Handle containing internal data.

Output Parameters

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

Inspector-Executor Sparse BLAS Execution Routines

Execution Routines and Their Data Types

Routine or Function Group	Data Types	Description
mkl_sparse_?_lu_smoother	s, d, c, z	Computes an action of a preconditioner which corresponds to the approximate matrix decomposition A ≈ (L+D)*E*(U+D) for the system Ax = b
mkl_sparse_?_mv	s, d, c, z	Computes a sparse matrix-vector product.
mkl_sparse_?_trsv	s, d, c, z	Solves a system of linear equations for a square sparse matrix.
mkl_sparse_?_mm	s, d, c, z	Computes the product of a sparse matrix and a dense matrix and stores the result as a dense matrix.
mkl_sparse_?_trsm	s, d, c, z	Solves a system of linear equations with multiple right-hand sides for a square sparse matrix.
mkl_sparse_?_add	s, d, c, z	Computes the sum of two sparse matrices. The result is stored in a newly allocated sparse matrix.
mkl_sparse_spmm	s, d, c, z	Computes the product of two sparse matrices and stores the result in a newly allocated sparse matrix.
mkl_sparse_?_spmmd	s, d, c, z	Computes the product of two sparse matrices and stores the result as a dense matrix.
mkl_sparse_sp2m	s, d, c, z	Computes the product of two sparse matrices (support operations on both matrices) and stores the result in a newly allocated sparse matrix.
mkl_sparse_?_sp2md	s, d, c, z	Computes the product of two sparse matrices (support operations on both matrices) and stores the result as a dense matrix.
mkl_sparse_sypr	s, d, c, z	Computes the symmetric product of three sparse matrices and stores the result in a newly allocated sparse matrix.
mkl_sparse_?_syprd	s, d, c, z	Computes the symmetric triple product of a sparse matrix and a dense matrix and stores the result as a dense matrix.

Routine or Function Group	Data Types	Description
mkl_sparse_?_symgs	s, d, c, z	Computes an action of a symmetric Gauss-Seidel preconditioner.
mkl_sparse_?_symgs_mv	s, d, c, z	Computes an action of a symmetric Gauss-Seidel preconditioner followed by a matrix-vector multiplication at the end.
mkl_sparse_?_syrkd	s, d, c, z	Computes the product of sparse matrix with its transpose (or conjugate transpose) and stores the result as a dense matrix.
mkl_sparse_syrk	s, d, c, z	Computes the product of a sparse matrix with its transpose (or conjugate transpose) and stores the result in a newly allocated sparse matrix.
mkl_sparse_?_dotmv	s, d, c, z	Computes a sparse matrix-vector product followed by a dot product.

mkl_sparse_?_lu_smoother

Computes an action of a preconditioner which corresponds to the approximate matrix decomposition $A \approx (L+D) \times E \times (U+D)$ *for the system* $Ax = b$ *(see description below).*

Syntax

```
status = mkl_sparse_s_lu_smoother (op, A, indx, descr, diag, approx_diag_inverse, x, b)
status = mkl_sparse_d_lu_smoother (op, A, indx, descr, diag, approx_diag_inverse, x, b)
status = mkl_sparse_c_lu_smoother (op, A, indx, descr, diag, approx_diag_inverse, x, b)
status = mkl_sparse_z_lu_smoother (op, A, indx, descr, diag, approx_diag_inverse, x, b)
```

Include Files

- mkl_spblas.f90

Description

This routine computes an update for an iterative solution *x* of the system *Ax=b* by means of applying one iteration of an approximate preconditioner which is based on the following approximation:

$A \sim (L+D) * E * (U+D)$, where *E* is an approximate inverse of the diagonal (using exact inverse will result in Gauss-Seidel preconditioner), *L* and *U* are lower/upper triangular parts of *A*, *D* is the diagonal (block diagonal in case of BSR format) of *A*.

The mkl_sparse_?_lu_smoother routine performs these operations:

```
r = b - A*x    /* 1. Computes the residual */
(L + D)*E*(U + D)*dx = r    /* 2. Finds the update dx by solving the system */
y = x + dx    /* 3. Performs an update */
```

This is also equal to the Symmetric Gauss-Seidel operation in the case of a CSR format and 1x1 diagonal blocks:

```
(L + D)*x^1 = b - U*x  /* Lower solve for intermediate x^1 */
(U + D)*x = b - L*x^1  /* Upper solve */
```

NOTE
This routine is supported only for non-transpose operation, real data types, and CSR/BSR sparse formats. In a BSR format, both diagonal values and approximate diagonal inverse arrays should be passed explicitly. For CSR format, diagonal values should be passed explicitly.

Input Parameters

operation

C_INT .

Specifies the operation performed on matrix *A*.

SPARSE_OPERATION_NON_TRANSPOSE, op(A) := A	**NOTE** Transpose and conjugate transpose (SPARSE_OPERATION_TRANSPOSE and SPARSE_OPERATION_CONJUGATE_TRANSPOSE) are not supported.
	Non-transpose, op(*A*) = *A*.

A

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

descr

MATRIX_DESCR.

Structure specifying sparse matrix properties.

sparse_matrix_type_t*type* - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only the requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

sparse_fill_mode_t*mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

sparse_diag_type_t*diag* - Specifies the diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT	Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT	Diagonal elements are equal to one.

NOTE
Only SPARSE_MATRIX_TYPE_GENERAL is supported.

diag

C_FLOAT for mkl_sparse_s_lu_smoother

C_DOUBLE for mkl_sparse_d_lu_smoother

C_FLOAT_COMPLEX for mkl_sparse_c_lu_smoother

C_DOUBLE_COMPLEX for mkl_sparse_z_lu_smoother

Array of size at least *m*, where *m* is the number of rows (or nrows * block_size * block_size in case of BSR format) of matrix *A*.

The array *diag* must contain the diagonal values of matrix *A*.

approx_diag_inverse

C_FLOAT for mkl_sparse_s_lu_smoother

C_DOUBLE for mkl_sparse_d_lu_smoother

C_FLOAT_COMPLEX for mkl_sparse_c_lu_smoother

C_DOUBLE_COMPLEX for mkl_sparse_z_lu_smoother

Array of size at least *m*, where *m* is the number of rows (or the number of rows * block_size * block_size in case of BSR format) of matrix *A*.

The array *approx_diag_inverse* will be used as *E*, approximate inverse of the diagonal of the matrix *A*.

x

C_FLOAT for mkl_sparse_s_lu_smoother

C_DOUBLE for mkl_sparse_d_lu_smoother

C_FLOAT_COMPLEX for mkl_sparse_c_lu_smoother

C_DOUBLE_COMPLEX for mkl_sparse_z_lu_smoother

Array of size at least *k*, where *k* is the number of columns (or columns * block_size in case of BSR format) of matrix *A*.

On entry, the array *x* must contain the input vector.

b

C_FLOAT for mkl_sparse_s_lu_smoother

C_DOUBLE for mkl_sparse_d_lu_smoother

C_FLOAT_COMPLEX for mkl_sparse_c_lu_smoother

C_DOUBLE_COMPLEX for mkl_sparse_z_lu_smoother

Array of size at least *m*, where *m* is the number of rows (or rows * block_size in case of BSR format) of matrix *A*. The array *b* must contain the values of the right-hand side of the system.

Output Parameters

x

`C_FLOAT` for `mkl_sparse_s_lu_smoother`

`C_DOUBLE` for `mkl_sparse_d_lu_smoother`

`C_FLOAT_COMPLEX` for `mkl_sparse_c_lu_smoother`

`C_DOUBLE_COMPLEX` for `mkl_sparse_z_lu_smoother`

Overwritten by the computed vector *y*.

status

`INTEGER`

Value indicating whether the operation was successful or not, and why.

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.
`SPARSE_STATUS_EXECUTION_FAILED`	Execution failed.
`SPARSE_STATUS_INTERNAL_ERROR`	An error in algorithm implementation occurred.
`SPARSE_STATUS_NOT_SUPPORTED`	The requested operation is not supported.

mkl_sparse_?_mv

Computes a sparse matrix- vector product.

Syntax

```
stat = mkl_sparse_s_mv (operation, alpha, A, descr, x, beta, y)
stat = mkl_sparse_d_mv (operation, alpha, A, descr, x, beta, y)
stat = mkl_sparse_c_mv (operation, alpha, A, descr, x, beta, y)
stat = mkl_sparse_z_mv (operation, alpha, A, descr, x, beta, y)
```

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_?_mv` routine computes a sparse matrix-dense vector product defined as

```
y := alpha*op(A)*x + beta*y
```

where:

alpha and *beta* are scalars, *x* and *y* are vectors, and *A* is a sparse matrix handle of a matrix with *m* rows and *k* columns, and `op` is a matrix modifier for matrix *A*.

Input Parameters

operation

`C_INT`.

Specifies operation `op()` on input matrix.

`SPARSE_OPERATION_NON_TRANSPOSE`	Non-transpose, `op(A) = A`.
`SPARSE_OPERATION_TRANSPOSE`	Transpose, `op(A) =` A^T.
`SPARSE_OPERATION_CONJUGATE_TRANSPOSE`	Conjugate transpose, `op(A) =` A^H.

alpha

`C_FLOAT` for `mkl_sparse_s_mv`

`C_DOUBLE` for `mkl_sparse_d_mv`

`C_FLOAT_COMPLEX` for `mkl_sparse_c_mv`

`C_DOUBLE_COMPLEX` for `mkl_sparse_z_mv`

Specifies the scalar *alpha*.

A

`SPARSE_MATRIX_T`.

Handle which contains the input matrix *A*.

descr

`MATRIX_DESCR`.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

`SPARSE_MATRIX_TYPE_GENERAL`	The matrix is processed as is.
`SPARSE_MATRIX_TYPE_SYMMETRIC`	The matrix is symmetric (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_HERMITIAN`	The matrix is Hermitian (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_TRIANGULAR`	The matrix is triangular (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_DIAGONAL`	The matrix is diagonal (only diagonal elements are processed).
`SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR`	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
`SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL`	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

	`SPARSE_FILL_MODE_LOWER` The lower triangular matrix part is processed. `SPARSE_FILL_MODE_UPPER` The upper triangular matrix part is processed. *diag* - Specifies diagonal type for non-general matrices: `SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one. `SPARSE_DIAG_UNIT` Diagonal elements are equal to one.

x	`C_FLOAT` for `mkl_sparse_s_mv`
	`C_DOUBLE` for `mkl_sparse_d_mv`
	`C_FLOAT_COMPLEX` for `mkl_sparse_c_mv`
	`C_DOUBLE_COMPLEX` for `mkl_sparse_z_mv`
	Array of size equal to the number of columns, *k* of *A* if *operation* = `SPARSE_OPERATION_NON_TRANSPOSE` and at least the number of rows, *m*, of *A* otherwise. On entry, the array must contain the vector *x*.

beta	`C_FLOAT` for `mkl_sparse_s_mv`
	`C_DOUBLE` for `mkl_sparse_d_mv`
	`C_FLOAT_COMPLEX` for `mkl_sparse_c_mv`
	`C_DOUBLE_COMPLEX` for `mkl_sparse_z_mv`
	Specifies the scalar *beta*.

y	`C_FLOAT` for `mkl_sparse_s_mv`
	`C_DOUBLE` for `mkl_sparse_d_mv`
	`C_FLOAT_COMPLEX` for `mkl_sparse_c_mv`
	`C_DOUBLE_COMPLEX` for `mkl_sparse_z_mv`
	Array with size at least *m* if *operation*=`SPARSE_OPERATION_NON_TRANSPOSE` and at least *k* otherwise. On entry, the array *y* must contain the vector *y*. Array of size equal to the number of rows, *m* of *A* if *operation* = `SPARSE_OPERATION_NON_TRANSPOSE` and at least the number of columns, *k*, of *A* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y	`C_FLOAT` for `mkl_sparse_s_mv`
	`C_DOUBLE` for `mkl_sparse_d_mv`
	`C_FLOAT_COMPLEX` for `mkl_sparse_c_mv`
	`C_DOUBLE_COMPLEX` for `mkl_sparse_z_mv`
	Overwritten by the updated vector *y*.

stat	`INTEGER` Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_trsv

Solves a system of linear equations for a triangular sparse matrix.

Syntax

```
stat = mkl_sparse_s_trsv (operation, alpha, A, descr, x, y)
stat = mkl_sparse_d_trsv (operation, alpha, A, descr, x, y)
stat = mkl_sparse_c_trsv (operation, alpha, A, descr, x, y)
stat = mkl_sparse_z_trsv (operation, alpha, A, descr, x, y)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_trsv routine solves a system of linear equations for a matrix:

```
op(A)*y = alpha * x
```

where *A* is a triangular sparse matrix , op is a matrix modifier for matrix *A*, *alpha* is a scalar, and *x* and *y* are vectors .

NOTE
For sparse matrices in the BSR format, the supported combinations of (*indexing*,*block_layout*) are:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

operation	C_INT. Specifies operation op() on input matrix.

	`SPARSE_OPERATION_NON_TRANSPOSE` Non-transpose, op(*A*) = *A*. `SPARSE_OPERATION_TRANSPOSE` Transpose, op(*A*) = A^T. `SPARSE_OPERATION_CONJUGATE_TRANSPOSE` Conjugate transpose, op(*A*) = A^H.

alpha	`C_FLOAT` for `mkl_sparse_s_trsv`
	`C_DOUBLE` for `mkl_sparse_d_trsv`
	`C_FLOAT_COMPLEX` for `mkl_sparse_c_trsv`
	`C_DOUBLE_COMPLEX` for `mkl_sparse_z_trsv`
	Specifies the scalar *alpha*.

A	`SPARSE_MATRIX_T`. Handle which contains the input matrix *A*.
descr	`MATRIX_DESCR`. Descriptor specifying sparse matrix properties. *type* - Specifies the type of a sparse matrix: `SPARSE_MATRIX_TYPE_GENERAL` The matrix is processed as is. `SPARSE_MATRIX_TYPE_SYMMETRIC` The matrix is symmetric (only the requested triangle is processed). `SPARSE_MATRIX_TYPE_HERMITIAN` The matrix is Hermitian (only the requested triangle is processed). `SPARSE_MATRIX_TYPE_TRIANGULAR` The matrix is triangular (only the requested triangle is processed). `SPARSE_MATRIX_TYPE_DIAGONAL` The matrix is diagonal (only diagonal elements are processed). `SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR` The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only. `SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL` The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only. *mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices: `SPARSE_FILL_MODE_LOWER` The lower triangular matrix part is processed. `SPARSE_FILL_MODE_UPPER` The upper triangular matrix part is processed. *diag* - Specifies diagonal type for non-general matrices:

	`SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one. `SPARSE_DIAG_UNIT` Diagonal elements are equal to one.

x	`C_FLOAT` for `mkl_sparse_s_trsv`
	`C_DOUBLE` for `mkl_sparse_d_trsv`
	`C_FLOAT_COMPLEX` for `mkl_sparse_c_trsv`
	`C_DOUBLE_COMPLEX` for `mkl_sparse_z_trsv`
	Array of size at least *m*, where *m* is the number of rows of matrix *A*. On entry, the array must contain the vector *x*.

Output Parameters

y	`C_FLOAT` for `mkl_sparse_s_trsv`
	`C_DOUBLE` for `mkl_sparse_d_trsv`
	`C_FLOAT_COMPLEX` for `mkl_sparse_c_trsv`
	`C_DOUBLE_COMPLEX` for `mkl_sparse_z_trsv`
	Array of size at least *m* containing the solution to the system of linear equations.

stat	`INTEGER` Value indicating whether the operation was successful or not, and why: `SPARSE_STATUS_SUCCESS` The operation was successful. `SPARSE_STATUS_NOT_INITIALIZED` The routine encountered an empty handle or matrix array. `SPARSE_STATUS_ALLOC_FAILED` Internal memory allocation failed. `SPARSE_STATUS_INVALID_VALUE` The input parameters contain an invalid value. `SPARSE_STATUS_EXECUTION_FAILED` Execution failed. `SPARSE_STATUS_INTERNAL_ERROR` An error in algorithm implementation occurred. `SPARSE_STATUS_NOT_SUPPORTED` The requested operation is not supported.

mkl_sparse_?_mm

Computes the product of a sparse matrix and a dense matrix and stores the result as a dense matrix.

Syntax

```
stat = mkl_sparse_s_mm (operation, alpha, A, descr, layout, B, columns, ldb, beta, C, ldc)
stat = mkl_sparse_d_mm (operation, alpha, A, descr, layout, B, columns, ldb, beta, C, ldc)
```

```
stat = mkl_sparse_c_mm (operation, alpha, A, descr, layout, B, columns, ldb, beta, C, ldc)
stat = mkl_sparse_z_mm (operation, alpha, A, descr, layout, B, columns, ldb, beta, C, ldc)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_mm routine performs a matrix-matrix operation:

```
C := alpha*op(A)*B + beta*C
```

where *alpha* and *beta* are scalars, *A* is a sparse matrix, op is a matrix modifier for matrix *A*, and *B* and *C* are dense matrices.

The mkl_sparse_?_mm and mkl_sparse_?_trsm routines support these configurations:

	Column-major dense matrix: *layout* = SPARSE_LAYOUT_COLUMN_MAJOR	Row-major dense matrix: *layout* = SPARSE_LAYOUT_ROW_MAJOR
0-based sparse matrix: SPARSE_INDEX_BASE_ZERO	CSR BSR: general non-transposed matrix multiplication only	All formats
1-based sparse matrix: SPARSE_INDEX_BASE_ONE	All formats	CSR BSR: general non-transposed matrix multiplication only

NOTE
For sparse matrices in the BSR format, the supported combinations of (*indexing*,*block_layout*) are:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

operation

C_INT.

Specifies operation op() on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.
SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.

alpha

C_FLOAT for mkl_sparse_s_mm

C_DOUBLE for mkl_sparse_d_mm

`C_FLOAT_COMPLEX` for `mkl_sparse_c_mm`

`C_DOUBLE_COMPLEX` for `mkl_sparse_z_mm`

Specifies the scalar *alpha*.

A

`SPARSE_MATRIX_T`.

Handle which contains the sparse matrix *A*.

descr

`MATRIX_DESCR`.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

`SPARSE_MATRIX_TYPE_GENERAL`	The matrix is processed as is.
`SPARSE_MATRIX_TYPE_SYMMETRIC`	The matrix is symmetric (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_HERMITIAN`	The matrix is Hermitian (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_TRIANGULAR`	The matrix is triangular (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_DIAGONAL`	The matrix is diagonal (only diagonal elements are processed).
`SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR`	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
`SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL`	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

`SPARSE_FILL_MODE_LOWER`	The lower triangular matrix part is processed.
`SPARSE_FILL_MODE_UPPER`	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

`SPARSE_DIAG_NON_UNIT`	Diagonal elements might not be equal to one.
`SPARSE_DIAG_UNIT`	Diagonal elements are equal to one.

layout

`C_INT`.

Describes the storage scheme for the dense matrix:

`SPARSE_LAYOUT_COLUMN_MAJOR`	Storage of elements uses column major layout.
`SPARSE_LAYOUT_ROW_MAJOR`	Storage of elements uses row major layout.

B

C_FLOAT for mkl_sparse_s_mm

C_DOUBLE for mkl_sparse_d_mm

C_FLOAT_COMPLEX for mkl_sparse_c_mm

C_DOUBLE_COMPLEX for mkl_sparse_z_mm

Array of size at least *rows***cols*.

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR
rows (number of rows in *B*)	*ldb*	If op(*A*) = *A*, number of columns in A If op(*A*) = A^T, number of rows in A
cols (number of columns in *B*)	*columns*	*ldb*

columns

C_INT.

Number of columns of matrix *C*.

ldb

C_INT.

Specifies the leading dimension of matrix *B*.

beta

C_FLOAT for mkl_sparse_s_mm

C_DOUBLE for mkl_sparse_d_mm

C_FLOAT_COMPLEX for mkl_sparse_c_mm

C_DOUBLE_COMPLEX for mkl_sparse_z_mm

Specifies the scalar *beta*

C

C_FLOAT for mkl_sparse_s_mm

C_DOUBLE for mkl_sparse_d_mm

C_FLOAT_COMPLEX for mkl_sparse_c_mm

C_DOUBLE_COMPLEX for mkl_sparse_z_mm

Array of size at least *rows***cols*, where

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR
rows (number of rows in *C*)	*ldc*	If op(*A*) = *A*, number of rows in A If op(*A*) = A^T, number of columns in A
cols (number of columns in *C*)	*columns*	*ldc*

ldc	Specifies the leading dimension of matrix *C*.

Output Parameters

C	C_FLOAT for mkl_sparse_s_mm C_DOUBLE for mkl_sparse_d_mm C_FLOAT_COMPLEX for mkl_sparse_c_mm C_DOUBLE_COMPLEX for mkl_sparse_z_mm Overwritten by the updated matrix *C*.
stat	INTEGER Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_trsm

Solves a system of linear equations with multiple right hand sides for a triangular sparse matrix.

Syntax

```
stat = mkl_sparse_s_trsm (operation, alpha, A, descr, layout, x, columns, ldx, y, ldy)
stat = mkl_sparse_d_trsm (operation, alpha, A, descr, layout, x, columns, ldx, y, ldy)
stat = mkl_sparse_c_trsm (operation, alpha, A, descr, layout, x, columns, ldx, y, ldy)
stat = mkl_sparse_z_trsm (operation, alpha, A, descr, layout, x, columns, ldx, y, ldy)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_trsm routine solves a system of linear equations with multiple right hand sides for a triangular sparse matrix:

```
Y := alpha*inv(op(A))*X
```

where:

alpha is a scalar, *X* and *Y* are dense matrices, *A* is a sparse matrix, and `op` is a matrix modifier for matrix *A*.

The `mkl_sparse_?_mm` and `mkl_sparse_?_trsm` routines support these configurations:

	Column-major dense matrix: *layout* = SPARSE_LAYOUT_COLUMN_MAJOR	Row-major dense matrix: *layout* = SPARSE_LAYOUT_ROW_MAJOR
0-based sparse matrix: SPARSE_INDEX_BASE_ZERO	CSR BSR: general non-transposed matrix multiplication only	All formats
1-based sparse matrix: SPARSE_INDEX_BASE_ONE	All formats	CSR BSR: general non-transposed matrix multiplication only

NOTE
For sparse matrices in the BSR format, the supported combinations of (*indexing*,*block_layout*) are:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

operation

C_INT.

Specifies operation `op()` on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.
SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.

alpha

C_FLOAT for `mkl_sparse_s_trsm`

C_DOUBLE for `mkl_sparse_d_trsm`

C_FLOAT_COMPLEX for `mkl_sparse_c_trsm`

C_DOUBLE_COMPLEX for `mkl_sparse_z_trsm`

Specifies the scalar *alpha*.

A

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

descr

MATRIX_DESCR.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.

SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT	Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT	Diagonal elements are equal to one.

layout

C_INT.

Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements uses column major layout.
SPARSE_LAYOUT_ROW_MAJOR	Storage of elements uses row major layout.

x

C_FLOAT for mkl_sparse_s_trsm

C_DOUBLE for mkl_sparse_d_trsm

C_FLOAT_COMPLEX for mkl_sparse_c_trsm

C_DOUBLE_COMPLEX for mkl_sparse_z_trsm

Array of size at least *rows**cols*.

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR
rows (number of rows in *x*)	*ldx*	number of rows in A

cols (number of columns in *x*)	*columns*	*ldx*

On entry, the array *x* must contain the matrix *X*.

columns

C_INT.

Number of columns in matrix *Y*.

ldx

C_INT.

Specifies the leading dimension of matrix *X*.

y

C_FLOAT for mkl_sparse_s_trsm

C_DOUBLE for mkl_sparse_d_trsm

C_FLOAT_COMPLEX for mkl_sparse_c_trsm

C_DOUBLE_COMPLEX for mkl_sparse_z_trsm

Array of size at least *rows***cols*, where

	layout = SPARSE_LAYOUT_COLUMN_MAJOR	*layout* = SPARSE_LAYOUT_ROW_MAJOR
rows (number of rows in *y*)	*ldy*	number of rows in A
cols (number of columns in *y*)	*columns*	*ldy*

Output Parameters

y

C_FLOAT for mkl_sparse_s_trsm

C_DOUBLE for mkl_sparse_d_trsm

C_FLOAT_COMPLEX for mkl_sparse_c_trsm

C_DOUBLE_COMPLEX for mkl_sparse_z_trsm

Overwritten by the updated matrix *Y*.

stat

INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.

SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_add

Computes the sum of two sparse matrices. The result is stored in a newly allocated sparse matrix.

Syntax

```
stat = mkl_sparse_s_add (operation, A, alpha, B, C)
stat = mkl_sparse_d_add (operation, A, alpha, B, C)
stat = mkl_sparse_c_add (operation, A, alpha, B, C)
stat = mkl_sparse_z_add (operation, A, alpha, B, C)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_add routine performs a matrix-matrix operation:

```
C := alpha*op(A) + B
```

where *alpha* is a scalar, *op* is a matrix modifier, and *A*, *B*, and *C* are sparse matrices.

NOTE
This routine is only supported for sparse matrices in CSR and BSR formats. It is not supported for COO or CSC formats.

Input Parameters

A

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

alpha

C_FLOAT for mkl_sparse_s_add

C_DOUBLE for mkl_sparse_d_add

C_FLOAT_COMPLEX for mkl_sparse_c_add

C_DOUBLE_COMPLEX for mkl_sparse_z_add

Specifies the scalar *alpha*.

operation

C_INT.

Specifies operation op() on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.
SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.

	SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.
B	SPARSE_MATRIX_T.	
	Handle which contains the sparse matrix *B*.	

Output Parameters

C	SPARSE_MATRIX_T.	
	Handle which contains the resulting sparse matrix.	
stat	INTEGER	
	Value indicating whether the operation was successful or not, and why:	
	SPARSE_STATUS_SUCCESS	The operation was successful.
	SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
	SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
	SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
	SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
	SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
	SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_spmm

Computes the product of two sparse matrices. The result is stored in a newly allocated sparse matrix.

Syntax

```
stat = mkl_sparse_spmm (operation, A, B, C)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_spmm routine performs a matrix-matrix operation:

```
C := op(A) *B
```

where *A*, *B*, and *C* are sparse matrices and *op* is a matrix modifier for matrix *A*.

Notes

- This routine is supported only for sparse matrices in CSC, CSR, and BSR formats. It is not supported for sparse matrices in COO format.
- The column indices of the output matrix (if in CSR format) can appear unsorted due to the algorithm chosen internally. To ensure sorted column indices (if that is important), call mkl_sparse_order().

Input Parameters

operation C_INT.

Specifies operation op() on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.
SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.

A SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

B SPARSE_MATRIX_T.

Handle which contains the sparse matrix *B*.

Output Parameters

C SPARSE_MATRIX_T.

Handle which contains the resulting sparse matrix.

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_spmmd

Computes the product of two sparse matrices and stores the result as a dense matrix.

Syntax

```
stat = mkl_sparse_s_spmmd (operation, A, B, layout, C, ldc)
stat = mkl_sparse_d_spmmd (operation, A, B, layout, C, ldc)
stat = mkl_sparse_c_spmmd (operation, A, B, layout, C, ldc)
stat = mkl_sparse_z_spmmd (operation, A, B, layout, C, ldc)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_spmmd routine performs a matrix-matrix operation:

```
C := op(A)*B
```

where *A* and *B* are sparse matrices, *op* is a matrix modifier for matrix *A*, and *C* is a dense matrix.

> **NOTE**
> This routine is not supported for sparse matrices in the COO format. For sparse matrices in BSR format, these combinations of (*indexing*, *block_layout*) are supported:
>
> - (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
> - (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

operation

C_INT.

Specifies operation op() on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(*A*) = *A*.
SPARSE_OPERATION_TRANSPOSE	Transpose, op(*A*) = A^T.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(*A*) = A^H.

A

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

B

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *B*.

layout

C_INT.

Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements uses column major layout.

	`SPARSE_LAYOUT_ROW_MAJOR` Storage of elements uses row major layout.
ldC	`C_INT`. Leading dimension of matrix *C*.

Output Parameters

C	`C_FLOAT` for `mkl_sparse_s_spmmd`
	`C_DOUBLE` for `mkl_sparse_d_spmmd`
	`C_FLOAT_COMPLEX` for `mkl_sparse_c_spmmd`
	`C_DOUBLE_COMPLEX` for `mkl_sparse_z_spmmd`
	Resulting dense matrix.

stat	`INTEGER` Value indicating whether the operation was successful or not, and why:

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.
`SPARSE_STATUS_EXECUTION_FAILED`	Execution failed.
`SPARSE_STATUS_INTERNAL_ERROR`	An error in algorithm implementation occurred.
`SPARSE_STATUS_NOT_SUPPORTED`	The requested operation is not supported.

mkl_sparse_sp2m

Computes the product of two sparse matrices. The result is stored in a newly allocated sparse matrix.

Syntax

```
stat = mkl_sparse_sp2m (opA, descrA, A, opB, descrB, B, request, C)
```

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_sp2m` routine performs a matrix-matrix operation:

```
C := opA(A) *opB(B)
```

where *A*,`B`, and *C* are sparse matrices, *opA* and *opB* are matrix modifiers for matrices *A* and *B*, respectively.

NOTE
The column indices of the output matrix (if in CSR format) can appear unsorted due to the algorithm chosen internally. To ensure sorted column indices (if that is important), call mkl_sparse_order().

Input Parameters

opA

C_INT.

Specifies operation on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(A)=A
SPARSE_OPERATION_TRANSPOSE	Transpose, op(A)=A^T
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(A)=A^H

opB

C_INT.

Specifies operation on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(B)=B
SPARSE_OPERATION_TRANSPOSE	Transpose, op(B)=B^T
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(B)=B^H

descrA

MATRIX_DESCR.

Structure that specifies sparse matrix properties.

NOTE Currently, only SPARSE_MATRIX_TYPE_GENERAL is supported.

sparse_matrix_type_t*type* specifies the type of sparse matrix.

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.

SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.

sparse_fill_mode_t*mode* specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

SPARSE_FILL_MODE_LOWER	The lower triangular matrix is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix is processed.

sparse_diag_type_t*diag* specifies the type of diagonal for non-general matrices.

SPARSE_DIAG_NON_UNIT	Diagonal elements must not be equal to 1.
SPARSE_DIAG_UNIT	Diagonal elements are equal to 1.

descrB

C_INT.

Structure that specifies sparse matrix properties.

NOTE Currently, only SPARSE_MATRIX_TYPE_GENERAL is supported.

sparse_matrix_type_t*type* specifies the type of sparse matrix.

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.

sparse_fill_mode_t*mode* specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

SPARSE_FILL_MODE_LOWER	The lower triangular matrix is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix is processed.

`sparse_diag_type_t`*diag* specifies the type of diagonal for non-general matrices.

SPARSE_DIAG_NON_UNIT	Diagonal elements must not be equal to 1.
SPARSE_DIAG_UNIT	Diagonal elements are equal to 1.

A

`SPARSE_MATRIX_T`.

Handle which contains the sparse matrix *A*.

B

`SPARSE_MATRIX_T`.

Handle which contains the sparse matrix *B*.

request

`C_INT`.

Specifies whether the full computations are performed at once or using the two-stage algorithm. See Two-stage Algorithm for Inspector-executor Sparse BLAS Routines.

SPARSE_STAGE_NNZ_COUNT	Only `rowIndex` (BSR/CSR format) or `colIndex` (CSC format) array of the matrix is computed internally. The computation can be extracted to measure the memory required for full operation.
SPARSE_STAGE_FINALIZE_MULT_NO_VAL	Finalize computations of the matrix structure (values will not be computed). Use only after the call with SPARSE_STAGE_NNZ_COUNT parameter.
SPARSE_STAGE_FINALIZE_MULT	Finalize computation. Can also be used when the matrix structure remains unchanged and only values of the resulting matrix *C* need to be recomputed.
SPARSE_STAGE_FULL_MULT_NO_VAL	Perform computations of the matrix structure.
SPARSE_STAGE_FULL_MULT	Perform the entire computation in a single step.

Output Parameters

C

`SPARSE_MATRIX_T`.

Handle which contains the resulting sparse matrix.

stat

`INTEGER`.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_sp2md

Computes the product of two sparse matrices (support operations on both matrices) and stores the result as a dense matrix.

Syntax

```
stat = mkl_sparse_s_sp2md (transA, descrA, A, transB, descrB, B, alpha, beta, C, layout, ldc )
stat = mkl_sparse_d_sp2md (transA, descrA, A, transB, descrB, B, alpha, beta, C, layout, ldc )
stat = mkl_sparse_c_sp2md (transA, descrA, A, transB, descrB, B, alpha, beta, C, layout, ldc )
stat = mkl_sparse_z_sp2md (transA, descrA, A, transB, descrB, B, alpha, beta, C, layout, ldc )
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_sp2md routine performs a matrix-matrix operation:

```
C = alpha * opA(A) *opB(B) + beta*C
```

where *A* and *B* are sparse matrices, *opA* is a matrix modifier for matrix *A*, *opB* is a matrix modifier for matrix *B*, and *C* is a dense matrix, *alpha* and *beta* are scalars.

NOTE
This routine is not supported for sparse matrices in the COO format. For sparse matrices in BSR format, these combinations of (indexing, block_layout) are supported:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

transA

C_INT.

Specifies operation op() on the input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, op(A)=A
SPARSE_OPERATION_TRANSPOSE	Transpose, op(A)=A^T
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, op(A)=A^H

descrA

MATRIX_DESCR.

Structure that specifies the sparse matrix properties.

NOTE Currently, only SPARSE_MATRIX_TYPE_GENERAL is supported.

sparse_matrix_type_t*type* specifies the type of sparse matrix.

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.

sparse_fill_mode_t*mode* specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

SPARSE_FILL_MODE_LOWER	The lower triangular matrix is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix is processed.

sparse_diag_type_t*diag* specifies the type of diagonal for non-general matrices.

SPARSE_DIAG_NON_UNIT	Diagonal elements must not be equal to 1.
SPARSE_DIAG_UNIT	Diagonal elements are equal to 1.

A

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

transB

C_INT.

Specifies operation opB() on the input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, opB(*B*)=*B*.
SPARSE_OPERATION_TRANSPOSE	Transpose, opB(*B*)=B^T .
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, opB(*B*)=B^H .

descrB

MATRIX_DESCR.

Structure that specifies the sparse matrix properties.

NOTE
Currently, only SPARSE_MATRIX_TYPE_GENERAL is supported.

sparse_matrix_type_t*type* specifies the type of sparse matrix.

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.

sparse_fill_mode_t*mode* specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

SPARSE_FILL_MODE_LOWER	The lower triangular matrix is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix is processed.

sparse_diag_type_t*diag* specifies the type of diagonal for non-general matrices.

SPARSE_DIAG_NON_UNIT	Diagonal elements must not be equal to 1.
SPARSE_DIAG_UNIT	Diagonal elements are equal to 1.

B

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *B*.

alpha

C_FLOAT for mkl_sparse_s_sp2md.

C_DOUBLE for mkl_sparse_d_sp2md.

C_FLOAT_COMPLEX for mkl_sparse_c_sp2md.

C_DOUBLE_COMPLEX for mkl_sparse_z_sp2md.

Specifies the scalar alpha.

beta

C_FLOAT for mkl_sparse_s_sp2md.

C_DOUBLE for mkl_sparse_d_sp2md.

C_FLOAT_COMPLEX for mkl_sparse_c_sp2md.

C_DOUBLE_COMPLEX for mkl_sparse_z_sp2md.

Specifies the scalar beta.

layout

C_INT.

Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements uses column major layout.
SPARSE_LAYOUT_ROW_MAJOR	Storage of elements uses row major layout.

ldc

C_INT.

Leading dimension of matrix *C*.

Output Parameters

C

C_FLOAT for mkl_sparse_s_sp2md.

C_DOUBLE for mkl_sparse_d_sp2md.

C_FLOAT_COMPLEX for mkl_sparse_c_sp2md.

C_DOUBLE_COMPLEX for mkl_sparse_z_sp2md.

The resulting dense matrix.

stat INTEGER

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_sypr

Computes the symmetric product of three sparse matrices and stores the result in a newly allocated sparse matrix.

Syntax

`stat = mkl_sparse_sypr (operation, A, B, descrB, C, request)`

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_sypr` routine performs a multiplication of three sparse matrices that results in a symmetric or Hermitian matrix, *C*.

```
C:=A*B*opA(A)
```

or

```
C:=opA(A)*B*A
```

depending on the matrix modifier *operation*.

Here, *A*, *B*, and *C* are sparse matrices, where *A* has a general structure while *B* and *C* are symmetric (for real data types) or Hermitian (for complex data types) matrices. `op`*A* is the transpose (real data types) or conjugate transpose (complex data types) operator.

NOTE
This routine is not supported for sparse matrices in COO or CSC formats. This routine supports only CSR and BSR formats. In addition, it supports only the sorted CSR and sorted BSR formats for the input matrix. If the data is unsorted, call the mkl_sparse_order routine before either mkl_sparse_sypr or mkl_sparse_?_syprd.

Input Parameters

operation

`C_INT`.

Specifies operation on the input sparse matrices.

`SPARSE_OPERATION_NON_TRANSPOSE`	Non-transpose case. $C:=A*B*(A^T)$ for real precision $C:=A*B*(A^H)$ for complex precision.
`SPARSE_OPERATION_TRANSPOSE`	Transpose case. This is not supported for complex matrices. $C:=(A^T)*B*A$
`SPARSE_OPERATION_CONJUGATE_TRANSPOSE`	Conjugate transpose case. This is not supported for real matrices. $C:=(A^H)*B*A$

A

`SPARSE_MATRIX_T`.

Handle which contains the sparse matrix *A*.

B

`SPARSE_MATRIX_T`.

Handle which contains the sparse matrix *B*.

descrB

`MATRIX_DESCR`.

Structure specifying properties of the sparse matrix.

`sparse_matrix_type_t` *type* specifies the type of a sparse matrix

SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the specified triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the specified triangle is processed).

`sparse_fill_mode_t` *mode* specifies the triangular matrix part.

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

`sparse_diag_type_t` *diag* specifies the type of diagonal.

SPARSE_DIAG_NON_UNIT	Diagonal elements cannot be equal to one.

NOTE
This routine also supports $C=AA^{T,H}$ with these parameters:

`descrB.type`=SPARSE_MATRIX_TYPE_DIAGONAL

`descrB.diag`=SPARSE_DIAG_UNIT

In this case, you do not need to allocate structure B. Use the routine as a 2-stage version of mkl_sparse_syrk.

request

`INTEGER`.

Use this routine to specify if the computations should be performed in a single step or using the two-stage algorithm. See Two-stage Algorithm for Inspector-executor Sparse BLAS Routines for more information.

SPARSE_STAGE_NNZ_COUNT	Only `rowIndex` (BSR/CSR format) or `colIndex` (CSC format) array of the matrix is computed internally. The computation can be extracted to measure the memory required for full operation.
SPARSE_STAGE_FINALIZE_MULT_NO_VAL	Finalize computations of the matrix structure (values will not be computed). Use only after the call with SPARSE_STAGE_NNZ_COUNT parameter.
SPARSE_STAGE_FINALIZE_MULT	Finalize computation. Can be used after the call with the SPARSE_STAGE_NNZ_COUNT or SPARSE_STAGE_FINALIZE_MULT_NO_VAL. Can also be used when the matrix structure remains unchanged and only values of the resulting matrix *C* need to be recomputed.
SPARSE_STAGE_FULL_MULT_NO_VAL	Perform computations of the matrix structure.
SPARSE_STAGE_FULL_MULT	Perform the entire computation in a single step.

Output Parameters

C

`SPARSE_MATRIX_T`.

Handle which contains the resulting sparse matrix. Only the upper-triangular part of the matrix is computed.

stat

INTEGER

Value indicating whether the operation was successful, or the reason why it failed.

SPARSE_STATUS_SUCCESS	The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_syprd

Computes the symmetric triple product of a sparse matrix and a dense matrix and stores the result as a dense matrix.

Syntax

```
stat = mkl_sparse_s_syprd (operation, A, B, denselayoutB, ldb, alpha, beta, C, denselayoutC, ldc)
stat = mkl_sparse_d_syprd (operation, A, B, denselayoutB, ldb, alpha, beta, C, denselayoutC, ldc)
stat = mkl_sparse_c_syprd (operation, A, B, denselayoutB, ldb, alpha, beta, C, denselayoutC, ldc)
stat = mkl_sparse_z_syprd (operation, A, B, denselayoutB, ldb, alpha, beta, C, denselayoutC, ldc)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_syprd routine performs a multiplication of three sparse matrices that results in a symmetric or Hermitian matrix, *C*.

```
C:=alpha*A*B*op(A) + beta*C
```

or

```
C:=alpha*op(A)*B*A + beta*C
```

depending on the matrix modifier operation. Here *A* is a sparse matrix, *B* and *C* are dense and symmetric (or Hermitian) matrices.

op is the transpose (real precision) or conjugate transpose (complex precision) operator.

NOTE
This routine is not supported for sparse matrices in COO or CSC formats. It supports only CSR and BSR formats. In addition, this routine supports only the sorted CSR and sorted BSR formats for the input matrix. If the data is unsorted, call the mkl_sparse_order routine before either mkl_sparse_sypr or mkl_sparse_?_syprd.

Input Parameters

operation

C_INT.

Specifies operation on the input sparse matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose case. $C := alpha*A*B*(A^T) + beta*C$ for real precision. $C := alpha*A*B*(A^H) + beta*C$ for complex precision.
SPARSE_OPERATION_TRANSPOSE	Transpose case. This is not supported for complex matrices. $C := alpha*(A^T)*B*A + beta*C$
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose case. This is not supported for real matrices. $C := alpha*(A^H)*B*A + beta*C$

A

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

B

SPARSE_MATRIX_T.

Input dense matrix. Only the upper triangular part of the matrix is used for computation.

denselayoutB

C_INT.

Structure that describes the storage scheme for the dense matrix.

SPARSE_LAYOUT_COLUMN_MAJOR	Store elements in a column-major layout.
SPARSE_LAYOUT_ROW_MAJOR	Store elements in a row-major layout.

ldb

SPARSE_MATRIX_T.

Leading dimension of matrix B.

alpha

Scalar parameter.

mkl_sparse_s_syprd	C_FLOAT
mkl_sparse_d_syprd	C_DOUBLE
mkl_sparse_c_syprd	C_FLOAT_COMPLEX
mkl_sparse_z_syprd	C_DOUBLE_COMPLEX

beta

Scalar parameter.

mkl_sparse_s_syprd	C_FLOAT
mkl_sparse_d_syprd	C_DOUBLE
mkl_sparse_c_syprd	C_FLOAT_COMPLEX
mkl_sparse_z_syprd	C_DOUBLE_COMPLEX

NOTE
Since the upper triangular part of matrix *C* is the only portion that is processed, set real values of `alpha` and `beta` in the complex case to obtain the Hermitian matrix.

denselayoutC

`C_INT`.

Structure that describes the storage scheme for the dense matrix.

SPARSE_LAYOUT_COLUMN_MAJOR	Store elements in a column-major layout.
SPARSE_LAYOUT_ROW_MAJOR	Store elements in a row-major layout.

ldc

`C_INT`.

Leading dimension of matrix C.

Output Parameters

C

`SPARSE_MATRIX_T`.

Handle which contains the resulting dense matrix. Only the upper-triangular part of the matrix is computed.

stat

INTEGER

Value indicating whether the operation was successful, or the reason why it failed.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_symgs

Computes a symmetric Gauss-Seidel preconditioner.

Syntax

```
stat = mkl_sparse_s_symgs (operation, A, descr, alpha, b, x)
stat = mkl_sparse_d_symgs (operation, A, descr, alpha, b, x)
stat = mkl_sparse_c_symgs (operation, A, descr, alpha, b, x)
stat = mkl_sparse_z_symgs (operation, A, descr, alpha, b, x)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_symgs routine performs this operation:

```
x0 := x*alpha;
(L + D)*x1 = b - U*x0;
(U + D)*x = b - L*x1;
```

where $A = L + D + U$.

NOTE
This routine is not supported for sparse matrices in BSR, COO, or CSC formats. It supports only the CSR format. Additionally, only symmetric matrices are supported, so the *desc.type* must be SPARSE_MATRIX_TYPE_SYMMETRIC.

Input Parameters

operation

C_INT.

Specifies the operation performed on matrix *A*.

SPARSE_OPERATION_NON_TRANSPOSE, op(*A*) := *A*.

NOTE
Transpose (SPARSE_OPERATION_TRANSPOSE) and conjugate transpose (SPARSE_OPERATION_CONJUGATE_TRANSPOSE) are not supported.

A

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

alpha

`C_FLOAT` for `mkl_sparse_s_symgs`

`C_DOUBLE` for `mkl_sparse_d_symgs`

`C_FLOAT_COMPLEX` for `mkl_sparse_c_symgs`

`C_DOUBLE_COMPLEX` for `mkl_sparse_z_symgs`

Specifies the scalar *alpha*.

descr

`MATRIX_DESCR`.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

`SPARSE_MATRIX_TYPE_GENERAL`	The matrix is processed as is.
`SPARSE_MATRIX_TYPE_SYMMETRIC`	The matrix is symmetric (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_HERMITIAN`	The matrix is Hermitian (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_TRIANGULAR`	The matrix is triangular (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_DIAGONAL`	The matrix is diagonal (only diagonal elements are processed).
`SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR`	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
`SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL`	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

`SPARSE_FILL_MODE_LOWER`	The lower triangular matrix part is processed.
`SPARSE_FILL_MODE_UPPER`	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

`SPARSE_DIAG_NON_UNIT`	Diagonal elements might not be equal to one.
`SPARSE_DIAG_UNIT`	Diagonal elements are equal to one.

x

`C_FLOAT` for `mkl_sparse_s_symgs`

`C_DOUBLE` for `mkl_sparse_d_symgs`

`C_FLOAT_COMPLEX` for `mkl_sparse_c_symgs`

`C_DOUBLE_COMPLEX` for `mkl_sparse_z_symgs`

Array of size at least *m*, where *m* is the number of rows of matrix *A*.

On entry, the array *x* must contain the vector *x*.

b

C_FLOAT for mkl_sparse_s_symgs

C_DOUBLE for mkl_sparse_d_symgs

C_FLOAT_COMPLEX for mkl_sparse_c_symgs

C_DOUBLE_COMPLEX for mkl_sparse_z_symgs

Array of size at least *m*, where *m* is the number of rows of matrix *A*.

On entry, the array *b* must contain the vector *b*.

Output Parameters

x Overwritten by the computed vector *x*.

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_symgs_mv

Computes a symmetric Gauss-Seidel preconditioner followed by a matrix-vector multiplication.

Syntax

```
stat = mkl_sparse_s_symgs_mv (operation, A, descr, alpha, b, x, y)
stat = mkl_sparse_d_symgs_mv (operation, A, descr, alpha, b, x, y)
stat = mkl_sparse_c_symgs_mv (operation, A, descr, alpha, b, x, y)
stat = mkl_sparse_z_symgs_mv (operation, A, descr, alpha, b, x, y)
```

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_?_symgs_mv` routine performs this operation:

```
x0 := x*alpha;
(L + D)*x1 = b - U*x0;
(U + D)*x = b - L*x1;
y := A*x
```

where `A = L + D + U`

NOTE
This routine is not supported for sparse matrices in BSR, COO, or CSC formats. It supports only the CSR format. Additionally, only symmetric matrices are supported, so the *desc.type* must be `SPARSE_MATRIX_TYPE_SYMMETRIC`.

Input Parameters

operation

`C_INT`.

Specifies the operation performed on input matrix.

`SPARSE_OPERATION_NON_TRANSPOSE`, `op(A) = A`.

NOTE
Transpose (`SPARSE_OPERATION_TRANSPOSE`) and conjugate transpose (`SPARSE_OPERATION_CONJUGATE_TRANSPOSE`) are not supported.

A

`SPARSE_MATRIX_T`.

Handle which contains the sparse matrix *A*.

alpha

`C_FLOAT` for `mkl_sparse_s_symgs_mv`

`C_DOUBLE` for `mkl_sparse_d_symgs_mv`

`C_FLOAT_COMPLEX` for `mkl_sparse_c_symgs_mv`

`C_DOUBLE_COMPLEX` for `mkl_sparse_z_symgs_mv`

Specifies the scalar *alpha*.

descr

`MATRIX_DESCR`.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

`SPARSE_MATRIX_TYPE_GENERAL`	The matrix is processed as is.
`SPARSE_MATRIX_TYPE_SYMMETRIC`	The matrix is symmetric (only the requested triangle is processed).
`SPARSE_MATRIX_TYPE_HERMITIAN`	The matrix is Hermitian (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT	Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT	Diagonal elements are equal to one.

x

C_FLOAT for mkl_sparse_s_symgs_mv

C_DOUBLE for mkl_sparse_d_symgs_mv

C_FLOAT_COMPLEX for mkl_sparse_c_symgs_mv

C_DOUBLE_COMPLEX for mkl_sparse_z_symgs_mv

Array of size at least *m*, where *m* is the number of rows of matrix *A*.

On entry, the array *x* must contain the vector *x*.

b

C_FLOAT for mkl_sparse_s_symgs_mv

C_DOUBLE for mkl_sparse_d_symgs_mv

C_FLOAT_COMPLEX for mkl_sparse_c_symgs_mv

C_DOUBLE_COMPLEX for mkl_sparse_z_symgs_mv

Array of size at least *m*, where *m* is the number of rows of matrix *A*.

On entry, the array *b* must contain the vector *b*.

Output Parameters

x

Overwritten by the computed vector *x*.

y

C_FLOAT for mkl_sparse_s_symgs_mv

C_DOUBLE for mkl_sparse_d_symgs_mv

C_FLOAT_COMPLEX for mkl_sparse_c_symgs_mv

C_DOUBLE_COMPLEX for mkl_sparse_z_symgs_mv

Array of size at least *m*, where *m* is the number of rows of matrix *A*.

Overwritten by the computed vector *y*.

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_syrk

Computes the product of sparse matrix with its transpose (or conjugate transpose) and stores the result in a newly allocated sparse matrix.

Syntax

```
stat = mkl_sparse_syrk (operation, A, C)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_syrk routine performs a sparse matrix-matrix operation which results in a sparse matrix C that is either Symmetric (real) or Hermitian (complex):

```
C := A*op(A)
```

where op(*) is the transpose for real matrices and conjugate transpose for complex matrices OR

```
C := op(A)*A
```

depending on the matrix modifier op which can be the transpose for real matrices or conjugate transpose for complex matrices.

Here, *A* and *C* are sparse matrices.

NOTE This routine is not supported for sparse matrices in COO or CSC formats. It supports only CSR and BSR formats. Additionally, this routine supports only the sorted CSR and sorted BSR formats for the input matrix. If data is unsorted, call the mkl_sparse_order routine before either mkl_sparse_syrk or mkl_sparse_?_syrkd.

Input Parameters

operation

Specifies the operation `op()` on input matrix .

`SPARSE_OPERATION_NON_TRANSPOSE`, Non-transpose, $C := A*op(A)$ where op(*) is the transpose for real matrices and conjugate transpose for complex matrices

`SPARSE_OPERATION_TRANSPOSE`, Transpose, $C := (A^T)*A$ for real matrix A

`SPARSE_OPERATION_CONJUGATE_TRANSPOSE`, Conjugate transpose, $C := (A^H)*A$ for complex matrix A.

A

`SPARSE_MATRIX_T`.

Handle which contains the sparse matrix *A*.

Output Parameters

C

`SPARSE_MATRIX_T`.

Handle which contains the resulting sparse matrix. Only the upper-triangular part of the matrix is computed.

stat

`INTEGER`

Value indicating whether the operation was successful or not, and why:

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.
`SPARSE_STATUS_EXECUTION_FAILED`	Execution failed.
`SPARSE_STATUS_INTERNAL_ERROR`	An error in algorithm implementation occurred.
`SPARSE_STATUS_NOT_SUPPORTED`	The requested operation is not supported.

mkl_sparse_?_syrkd

Computes the product of sparse matrix with its transpose (or conjugate transpose) and stores the result as a dense matrix.

Syntax

```
stat = mkl_sparse_s_syrkd (operation, A, alpha, beta, C, layout, ldc)
stat = mkl_sparse_d_syrkd (operation, A, alpha, beta, C, layout, ldc)
stat = mkl_sparse_c_syrkd (operation, A, alpha, beta, C, layout, ldc)
stat = mkl_sparse_z_syrkd (operation, A, alpha, beta, C, layout, ldc)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_syrkd routine performs a sparse matrix-matrix operation which results in a dense matrix C that is either symmetric (real case) or Hermitian (complex case):

```
C := beta*C + alpha*A*op(A)
```

or

```
C := beta*C + alpha*op(A)*A
```

depending on the matrix modifier op which can be the transpose for real matrices or conjugate transpose for complex matrices. Here, *A* is a sparse matrix and *C* is a dense matrix.

NOTE This routine is not supported for sparse matrices in COO or CSC formats. It supports only CSR and BSR formats. Additionally, this routine supports only the sorted CSR and sorted BSR formats for the input matrix. If data is unsorted, call the mkl_sparse_order routine before either mkl_sparse_syrk or mkl_sparse_?_syrkd.

Input Parameters

operation

C_INT.

Specifies the operation op() performed on the input matrix.

SPARSE_OPERATION_NON_TRANSPOSE, Non-transpose, $C := beta*C + alpha*A*op(A)$ where op(*) is the transpose (real matrices) or conjugate transpose (complex matrices).

SPARSE_OPERATION_TRANSPOSE, Transpose, $C := beta*C + alpha*A^{T}*A$ for real matrix A.

SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $C := beta*C + alpha*A^{H}*A$ for complex matrix A.

A

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

alpha

C_FLOAT for mkl_sparse_s_syrkd

C_DOUBLE for mkl_sparse_d_syrkd

C_FLOAT_COMPLEX for mkl_sparse_c_syrkd

C_DOUBLE_COMPLEX for mkl_sparse_z_syrkd

Scalar parameter *alpha*.

beta

C_FLOAT for mkl_sparse_s_syrkd

C_DOUBLE for mkl_sparse_d_syrkd

C_FLOAT_COMPLEX for mkl_sparse_c_syrkd

C_DOUBLE_COMPLEX for mkl_sparse_z_syrkd

Scalar parameter *beta*.

layout Describes the storage scheme for the dense matrix.

layout = SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements uses column-major layout.
layout = SPARSE_LAYOUT_ROW_MAJOR	Storage of elements uses row-major layout.

ldc C_INT.

Leading dimension of matrix *C*.

NOTE
Only the upper triangular part of matrix C is processed. Therefore, you must set real values of *alpha* and *beta* for complex matrices in order to obtain a Hermitian matrix.

Output Parameters

C SPARSE_MATRIX_T.

Resulting dense matrix. Only the upper triangular part of the matrix is computed.

stat INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_dotmv

Computes a sparse matrix-vector product followed by a dot product.

Syntax

```
stat = mkl_sparse_s_dotmv (operation, alpha, A, descr, x, beta, y, d)
stat = mkl_sparse_d_dotmv (operation, alpha, A, descr, x, beta, y, d)
stat = mkl_sparse_c_dotmv (operation, alpha, A, descr, x, beta, y, d)
stat = mkl_sparse_z_dotmv (operation, alpha, A, descr, x, beta, y, d)
```

Include Files

- mkl_spblas.f90

Description

The mkl_sparse_?_dotmv routine computes a sparse matrix-vector product and dot product:

```
y := alpha*op(A)*x + beta*yd := ∑ᵢxᵢ*yᵢ (real case)
d := ∑ᵢconj(xᵢ)*yᵢ (complex case)
```

where

- *alpha* and *beta* are scalars.
- *x* and *y* are vectors.
- *A* is an *m*-by-*k* matrix.
- *conj* represents complex conjugation.
- op(*A*) is a matrix modifier.

Available options for op(*A*) are *A*, A^T, or A^H.

> **NOTE**
> For sparse matrices in the BSR format, the supported combinations of (*indexing*,*block_layout*) are:
>
> - (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
> - (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

operation

C_INT.

Specifies the operation performed on matrix *A*.

If *operation* = SPARSE_OPERATION_NON_TRANSPOSE, op(*A*) = *A*.

If *operation* = SPARSE_OPERATION_TRANSPOSE, op(*A*) = A^T.

If *operation* = SPARSE_OPERATION_CONJUGATE_TRANSPOSE, op(*A*) = A^H.

alpha

C_FLOAT for mkl_sparse_s_dotmv

C_DOUBLE for mkl_sparse_d_dotmv

C_FLOAT_COMPLEX for mkl_sparse_c_dotmv

C_DOUBLE_COMPLEX for mkl_sparse_z_dotmv

Specifies the scalar *alpha*.

A

SPARSE_MATRIX_T.

Handle which contains the sparse matrix *A*.

descr

MATRIX_DESCR.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is.

SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT	Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT	Diagonal elements are equal to one.

x

C_FLOAT for mkl_sparse_s_dotmv

C_DOUBLE for mkl_sparse_d_dotmv

C_FLOAT_COMPLEX for mkl_sparse_c_dotmv

C_DOUBLE_COMPLEX for mkl_sparse_z_dotmv

If *operation* = SPARSE_OPERATION_NON_TRANSPOSE, array of size at least *k*, where *k* is the number of columns of matrix *A*.

Otherwise, array of size at least *m*, where *m* is the number of rows of matrix *A*.

On entry, the array *x* must contain the vector *x*.

beta

C_FLOAT for mkl_sparse_s_dotmv

C_DOUBLE for mkl_sparse_d_dotmv

C_FLOAT_COMPLEX for mkl_sparse_c_dotmv

C_DOUBLE_COMPLEX for mkl_sparse_z_dotmv

Specifies the scalar *beta*.

y

C_FLOAT for mkl_sparse_s_dotmv

C_DOUBLE for mkl_sparse_d_dotmv

C_FLOAT_COMPLEX for mkl_sparse_c_dotmv

C_DOUBLE_COMPLEX for mkl_sparse_z_dotmv

If *operation* = SPARSE_OPERATION_NON_TRANSPOSE, array of size at least *m*, where *k* is the number of rows of matrix *A*.

Otherwise, array of size at least *k*, where *k* is the number of columns of matrix *A*.

On entry, the array *y* must contain the vector *y*.

Output Parameters

y

Overwritten by the updated vector *y*.

d

C_FLOAT for mkl_sparse_s_dotmv

C_DOUBLE for mkl_sparse_d_dotmv

C_FLOAT_COMPLEX for mkl_sparse_c_dotmv

C_DOUBLE_COMPLEX for mkl_sparse_z_dotmv

Overwritten by the dot product of *x* and *y*.

stat

INTEGER

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_sorv

Computes forward, backward sweeps or a symmetric successive over-relaxation preconditioner operation.

Syntax

```
stat = sparse_status_t mkl_sparse_s_sorv(type, descrA, A, omega, alpha, x, b)
```

```
stat = sparse_status_t mkl_sparse_d_sorv(type, descrA, A, omega, alpha, x, b)
```

Include Files

- `mkl_spblas.f90`

Description

The mkl_sparse_?_sorv routine performs one of the following operations:

SPARSE_SOR_FORWARD:

$$(\omega*L+D)*x\text{^}1 = (D-\omega*D-\omega*U)*x\text{^}0 + \omega*b$$

SPARSE_SOR_BACKWARD:

$$(\omega*U+D)*x\text{^}1 = (D-\omega*D-\omega*L)*x\text{^}0 + \omega*b$$

SPARSE_SOR_SYMMETRIC: Performs application of a

$$\frac{\omega}{2-\omega}\left(\frac{1}{\omega}D+L\right)D^{-1}\left(\frac{1}{\omega}D+L\right)^{T}$$

preconditioner.

where A = L + D + U and x^0 is an input vector x scaled by input parameter alpha vector and x^1 is an output stored in vector x.

NOTE
Currently this routine only supports the following configuration:

- CSR format of the input matrix
- `SPARSE_SOR_FORWARD` operation
- General matrix (`descr.type` is `SPARSE_MATRIX_TYPE_GENERAL`) or symmetric matrix with full portrait and unit diagonal (`descr.type` is `SPARSE_MATRIX_TYPE_SYMMETRIC`, `descr.mode` is `SPARSE_FILL_MODE_FULL`, and `descr.diag` is `SPARSE_DIAG_UNIT`)

NOTE
Currently, this routine is optimized only for sequential threading execution mode.

Warning It is currently not allowed to place a `sorv` call in a parallel section (e.g., under `#pragma omp parallel`), because it is not thread-safe in this scenario. This limitation will be addressed in one of the upcoming releases.

Product and Performance Information
Notice revision #20201201

Input Parameters

type SPARSE_MATRIX_T.

Specifies the operation performed by the SORV preconditioner.

SPARSE_SOR_FORWARD Performs forward sweep as defined by:

$$(\omega*L+D)*x\text{^}1 = (D-\omega*D-\omega*U)*x\text{^}0 + \omega*b$$

SPARSE_SOR_BACKWARD Performs backward sweep as defined by:

$$(\omega*U+D)*x\text{^}1 = (D-\omega*D-\omega*L)*x\text{^}0 + \omega*b$$

SPARSE_SOR_SYMMETRIC Preconditioner matrix could be expressed as:

$$\frac{\omega}{2-\omega}\left(\frac{1}{\omega}D+L\right)D^{-1}\left(\frac{1}{\omega}D+L\right)^{T}$$

descr MATRIX_DESCR.

Structure specifying sparse matrix properties.

SPARSE_MATRIX_T type Specifies the type of a sparse matrix:

- SPARSE_MATRIX_TYPE_GENERAL

  The matrix is processed as-is.
- SPARSE_MATRIX_TYPE_SYMMETRIC

  The matrix is symmetric (only the requested triangle is processed).
- SPARSE_MATRIX_TYPE_HERMITIAN

  The matrix is Hermitian (only the requested triangle is processed).
- SPARSE_MATRIX_TYPE_TRIANGULAR

  The matrix is triangular (only the requested triangle is processed).
- SPARSE_MATRIX_TYPE_DIAGONAL

  The matrix is diagonal (only diagonal elements are processed).
- SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR

  The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
- SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL

  The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`C_INT mode`	Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices: • `SPARSE_FILL_MODE_LOWER` The lower triangular matrix part is processed. • `SPARSE_FILL_MODE_UPPER` The upper triangular matrix part is processed.
`SPARSE_MATRIX_TYPE_DIAGONAL diag`	Specifies diagonal type for non-general matrices: • `SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one. • `SPARSE_DIAG_UNIT` Diagonal elements are equal to one.

A

`SPARSE_MATRIX_T`.

Handle containing internal data.

omega

`C_FLOAT`.

Relaxation factor.

alpha

`C_FLOAT`.

Parameter that could be used to normalize or set to zero the vector x that holds the initial guess.

x

`C_FLOAT`.

Initial guess on input.

b

`C_FLOAT`.

Right-hand side.

Output Parameters

x

`C_FLOAT`.

Solution vector on output.

stat

`INTEGER`

Value indicating whether the operation was successful or not, and why:

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

BLAS-like Extensions

Intel® oneAPI Math Kernel Library provides C and Fortran routines to extend the functionality of the BLAS routines. These include routines to compute vector products, matrix-vector products, and matrix-matrix products.

Intel® oneAPI Math Kernel Library also provides routines to perform certain data manipulation, including matrix in-place and out-of-place transposition operations combined with simple matrix arithmetic operations. Transposition operations are Copy As Is, Conjugate transpose, Transpose, and Conjugate. Each routine adds the possibility of scaling during the transposition operation by giving some *alpha* and/or *beta* parameters. Each routine supports both row-major orderings and column-major orderings.

Table "BLAS-like Extensions" lists these routines.

The <?> symbol in the routine short names is a precision prefix that indicates the data type:

s	REAL
d	DOUBLE PRECISION
c	COMPLEX
z	DOUBLE COMPLEX

BLAS-like Extensions

Routine	Data Types	Description
?axpby	s, d, c, z	Scales two vectors, adds them to one another and stores result in the vector (routines).
?axpy_batch ?axpy_batch_strided	s, d, c, z	Computes groups of vector-scalar products added to a vector.
?dgmm_batch_strided ?dgmm_batch	s, d, c, z	Computes groups of diagonal matrix-general matrix product
?gem2vc	c, z	Two matrix-vector products using a general matrix, complex data.
?gem2vu	s, d	Two matrix-vector products using a general matrix, real data.
?gemm_batch ?gemm_batch_strided	s, d, c, z	Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.
gemm_*	Integer	Computes a matrix-matrix product with general integer matrices.

Routine	Data Types	Description
?gemm_compute	h, s, d	Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.
gemm_*_compute	Integer	Computes a matrix-matrix product with general integer matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.
?gemm_pack	h, s, d	Performs scaling and packing of the matrix into the previously allocated buffer.
gemm_*_pack	Integer,	Pack the matrix into the buffer allocated previously.
?gemm_pack_get_size	h, s, d	Returns the number of bytes required to store the packed matrix.
gemm_*_pack_get_size	Integer,	Returns the number of bytes required to store the packed matrix.
?gemm3m	c, z	Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.
?gemm3m_batch ?gemm3m_batch_strided	c, z	Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.
?gemmt	s, d, c, z	Computes a matrix-matrix product with general matrices but updates only the upper or lower triangular part of the result matrix.
?gemv_batch_strided ?gemv_batch	s, d, c, z	Computes groups of matrix-vector product using general matrices.
?trsm_batch ?trsm_batch_strided	s, d, c, z	Solves a triangular matrix equation for a group of matrices.
mkl_?imatcopy	s, d, c, z	Performs scaling and in-place transposition/copying of matrices.
mkl_?imatcopy_batch_strided mkl_?imatcopy_batch	s, d, c, z	Computes groups of in-place matrix copy/transposition with scaling using general matrices.
mkl_?omatadd	s, d, c, z	Performs scaling and sum of two matrices including their out-of-place transposition/copying.
mkl_?omatcopy	s, d, c, z	Performs scaling and out-of-place transposition/copying of matrices.
mkl_?omatcopy_batch_strided mkl_?omatcopy_batch	s, d, c, z	Computes groups of out of place matrix copy/transposition with scaling using general matrices.
mkl_?omatcopy2	s, d, c, z	Performs two-strided scaling and out-of-place transposition/copying of matrices.

Routine	Data Types	Description
mkl_jit_create_?gemm	s, d, c, z	Creates a handle on a jitter and generates a GEMM kernel that computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product, with general matrices.
mkl_jit_destroy		Deletes the previously created jitter and the generated GEMM kernel.
mkl_jit_get_?gemm_ptr	s, d, c, z	Returns the GEMM kernel previously generated.

?axpy_batch

Computes a group of vector-scalar products added to a vector.

Syntax

```
call saxpy_batch(n_array, alpha_array, x_array, incx_array, y_array, incy_array, group_count, group_size_array)
call daxpy_batch(n_array, alpha_array, x_array, incx_array, y_array, incy_array, group_count, group_size_array)
call caxpy_batch(n_array, alpha_array, x_array, incx_array, y_array, incy_array, group_count, group_size_array)
call zaxpy_batch(n_array, alpha_array, x_array, incx_array, y_array, incy_array, group_count, group_size_array)
```

Description

The ?axpy_batch routines perform a series of scalar-vector product added to a vector. They are similar to the ?axpy routine counterparts, but the ?axpy_batch routines perform vector operations with a group of vectors. The groups contain vectors with the same parameters.

The operation is defined as

```
idx = 0
for i = 0 … group_count – 1
    n, alpha, incx, incy and group_size at position i in n_array, alpha_array, incx_array,
incy_array and group_size_array
    for j = 0 … group_size – 1
        x and y are vectors of size n at position idx in x_array and y_array
        y := alpha * x + y
        idx := idx + 1
    end for
end for
```

The number of entries in *x_array*, and *y_array* is *total_batch_count* = the sum of all of the *group_size* entries.

Input Parameters

n_array

INTEGER. Array of size *group_count*. For the group i, n_i = n_array[i] is the number of elements in vectors *x* and *y*.

alpha_array

REAL for saxpy_batch

DOUBLE PRECISION for daxpy_batch

	COMPLEX for `caxpy_batch` DOUBLE COMPLEX for `zaxpy_batch` Array of size *group_count*. For the group `i`, $alpha_i$ = `alpha_array[i]` is the scalar *alpha*.
x_array	INTEGER*8 for Intel® 64 architecture INTEGER*4 for IA-32 architecture Array of size *total_batch_count* of pointers used to store *x* vectors. The array allocated for the *x* vectors of the group `i` must be of size at least (1 + (n_i – 1)*abs($incx_i$)).
incx_array	INTEGER. Array of size *group_count*. For the group `i`, $incx_i$ = `incx_array[i]` is the stride of vector *x*.
y_array	INTEGER*8 for Intel® 64 architecture INTEGER*4 for IA-32 architecture Array of size *total_batch_count* of pointers used to store *y* vectors. The array allocated for the *y* vectors of the group `i` must be of size at least (1 + (n_i – 1)*abs($incy_i$)).
incy_array	INTEGER. Array of size *group_count*. For the group `i`, $incy_i$ = `incy_array[i]` is the stride of vector *y*.
group_count	INTEGER. Number of groups. Must be at least 0.
group_size_array	INTEGER. Array of size *group_count*. The element `group_size_array[i]` is the number of vector in the group `i`. Each element in *group_size_array* must be at least 0.

Output Parameters

y_array	Array of pointers holding the *total_batch_count* updated vector *y*.

?axpy_batch_strided

Computes a group of vector-scalar products added to a vector.

Syntax

```
call saxpy_batch_strided(n, alpha, x, incx, stridex, y, incy, stridey, batch_size)
call daxpy_batch_strided(n, alpha, x, incx, stridex, y, incy, stridey, batch_size)
call caxpy_batch_strided(n, alpha, x, incx, stridex, y, incy, stridey, batch_size)
call zaxpy_batch_strided(n, alpha, x, incx, stridex, y, incy, stridey, batch_size)
```

Include Files

- `mkl.fi`

Description

The `?axpy_batch_strided` routines perform a series of scalar-vector product added to a vector. They are similar to the `?axpy` routine counterparts, but the `?axpy_batch_strided` routines perform vector operations with a group of vectors.

All vector *x* (respectively, *y*) have the same parameters (size, increments) and are stored at constant *stridex* (respectively, *stridey*) from each other. The operation is defined as

```
For i = 0 … batch_size – 1
    X and Y are vectors at offset i * stridex and i * stridey in x and y
    Y = alpha * X + Y
end for
```

Input Parameters

n	INTEGER. Number of elements in vectors *x* and *y*.
alpha	REAL for `saxpy_batch_strided` DOUBLE PRECISION for `daxpy_batch_strided` COMPLEX for `caxpy_batch_strided` DOUBLE COMPLEX for `zaxpy_batch_strided` Specifies the scalar *alpha*.
x	REAL for `saxpy_batch_strided` DOUBLE PRECISION for `daxpy_batch_strided` COMPLEX for `caxpy_batch_strided` DOUBLE COMPLEX for `zaxpy_batch_strided` Array of size at least *stridex***batch_size* holding the *x* vectors.
incx	INTEGER. Specifies the increment for the elements of *x*.
stridex	INTEGER. Stride between two consecutive *x* vectors; must be at least zero.
y	REAL for `saxpy_batch_strided` DOUBLE PRECISION for `daxpy_batch_strided` COMPLEX for `caxpy_batch_strided` DOUBLE COMPLEX for `zaxpy_batch_strided` Array of size at least *stridey***batch_size* holding the *y* vectors.
incy	INTEGER. Specifies the increment for the elements of *y*.
stridey	INTEGER. Stride between two consecutive *y* vectors; must be at least (1 + (n-1)*abs(*incy*)).
batch_size	INTEGER. Number of `axpy` computations to perform and *x* and *y* vectors. Must be at least 0.

Output Parameters

y	Array holding the *batch_size* updated vector *y*.

?axpby

Scales two vectors, adds them to one another and stores result in the vector.

Syntax

```
call saxpby(n, a, x, incx, b, y, incy)
call daxpby(n, a, x, incx, b, y, incy)
call caxpby(n, a, x, incx, b, y, incy)
call zaxpby(n, a, x, incx, b, y, incy)
call axpby(x, y [,a] [,b])
```

Include Files

- mkl.fi, blas.f90

Description

The ?axpby routines perform a vector-vector operation defined as

```
y := a*x + b*y
```

where:

a and *b* are scalars

x and *y* are vectors each with *n* elements.

Input Parameters

n	INTEGER. Specifies the number of elements in vectors *x* and *y*.
a	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby Specifies the scalar *a*.
x	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby Array, size at least (1 + (*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
b	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby Specifies the scalar *b*.
y	REAL for saxpby DOUBLE PRECISION for daxpby

	`COMPLEX` for `caxpby`
	`DOUBLE COMPLEX` for `zaxpby`
	Array, size at least `(1 + (n-1)*abs(incy))`.

incy	`INTEGER`. Specifies the increment for the elements of *y*.

Output Parameters

y	Contains the updated vector *y*.

Example

For examples of routine usage, see these code examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- `saxpby`: `examples\blas\source\saxpbyx.f`
- `daxpby`: `examples\blas\source\daxpbyx.f`
- `caxpby`: `examples\blas\source\caxpbyx.f`
- `zaxpby`: `examples\blas\source\zaxpbyx.f`

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `axpby` interface are the following:

x	Holds the array of size *n*.
y	Holds the array of size *n*.
a	The default value is 1.
b	The default value is 1.

?gem2vu

Computes two matrix-vector products using a general matrix (real data)

Syntax

```
call sgem2vu(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call dgem2vu(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call gem2vu(a, x1, x2, y1, y2 [,alpha][,beta] )
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?gem2vu` routines perform two matrix-vector operations defined as

*y1 := alpha*A*x1 + beta*y1*,

and

y2 := *alpha*A'*x2* + *beta*y2*,

where:

alpha and *beta* are scalars,

x1, *x2*, *y1*, and *y2* are vectors,

A is an *m*-by-*n* matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.
alpha	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Specifies the scalar *alpha*.
a	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Array, size (*lda*, *n*). Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.
lda	INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).
x1	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Array, size at least (1+(*n*–1)*abs(*incx1*)). Before entry, the incremented array *x1* must contain the vector *x1*.
incx1	INTEGER. Specifies the increment for the elements of *x1*. The value of *incx1* must not be zero.
x2	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Array, size at least (1+(*m*–1)*abs(*incx2*)). Before entry, the incremented array *x2* must contain the vector *x2*.
incx2	INTEGER. Specifies the increment for the elements of *x2*. The value of *incx2* must not be zero.
beta	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Specifies the scalar *beta*. When *beta* is set to zero, then *y1* and *y2* need not be set on input.
y1	REAL for sgem2vu DOUBLE PRECISION for dgem2vu

	Array, size at least `(1+(m-1)*abs(incy1))`. Before entry with non-zero *beta*, the incremented array *y1* must contain the vector *y1*.
incy1	`INTEGER`. Specifies the increment for the elements of *y1*. The value of *incy1* must not be zero.

y	`REAL` for `sgem2vu`
	`DOUBLE PRECISION` for `dgem2vu`
	Array, size at least `(1+(n-1)*abs(incy2))`. Before entry with non-zero *beta*, the incremented array *y2* must contain the vector *y2*.

incy2	`INTEGER`. Specifies the increment for the elements of *y2*. The value of *incy2* must not be zero.

Output Parameters

y1	Updated vector *y1*.
y2	Updated vector *y2*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `gem2vu` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
x1	Holds the vector with the number of elements *rx1* where *rx1* = *n*.
x2	Holds the vector with the number of elements *rx2* where *rx2* = *m*.
y1	Holds the vector with the number of elements *ry1* where *ry1* = *m*.
y2	Holds the vector with the number of elements *ry2* where *ry2* = *n*.
alpha	The default value is 1.
beta	The default value is 0.

?gem2vc

Computes two matrix-vector products using a general matrix (complex data)

Syntax

```
call cgem2vc(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call zgem2vc(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call gem2vc(a, x1, x2, y1, y2 [,alpha][,beta] )
```

Include Files

- `mkl.fi, blas.f90`

Description

The ?gem2vc routines perform two matrix-vector operations defined as

y1 := *alpha***A***x1* + *beta***y1*,

and

y2 := *alpha**conjg(*A*')**x2* + *beta***y2*,

where:

alpha and *beta* are scalars,

x1, *x2*, *y1*, and *y2* are vectors,

A is an *m*-by-*n* matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.
alpha	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Specifies the scalar *alpha*.
a	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Array, size (*lda*, *n*). Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.
lda	INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).
x1	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Array, size at least (1+(*n*–1)*abs(*incx1*)). Before entry, the incremented array *x1* must contain the vector *x1*.
incx1	INTEGER. Specifies the increment for the elements of *x1*. The value of *incx1* must not be zero.
x2	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Array, size at least (1+(*m*–1)*abs(*incx2*)). Before entry, the incremented array *x2* must contain the vector *x2*.
incx2	INTEGER. Specifies the increment for the elements of *x2*. The value of *incx2* must not be zero.

beta	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Specifies the scalar *beta*. When *beta* is set to zero, then *y1* and *y2* need not be set on input.
y1	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Array, size at least (1+(*m*-1)*abs(*incy1*)). Before entry with non-zero *beta*, the incremented array *y1* must contain the vector *y1*.
incy1	INTEGER. Specifies the increment for the elements of *y1*. The value of *incy1* must not be zero.
y2	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Array, size at least (1+(*n*-1)*abs(*incy2*)). Before entry with non-zero *beta*, the incremented array *y2* must contain the vector *y2*.
incy2	INTEGER. Specifies the increment for the elements of *y2*. The value of *incy* must not be zero. INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

y1	Updated vector *y1*.
y2	Updated vector *y2*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine gem2vc interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
x1	Holds the vector with the number of elements *rx1* where *rx1* = *n*.
x2	Holds the vector with the number of elements *rx2* where *rx2* = *m*.
y1	Holds the vector with the number of elements *ry1* where *ry1* = *m*.
y2	Holds the vector with the number of elements *ry2* where *ry2* = *n*.
alpha	The default value is 1.
beta	The default value is 0.

?gemmt

Computes a matrix-matrix product with general matrices but updates only the upper or lower triangular part of the result matrix.

Syntax

```
call sgemmt (uplo, transa, transb, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dgemmt (uplo, transa, transb, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call cgemmt (uplo, transa, transb, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemmt (uplo, transa, transb, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call gemmt (a, b, c[, uplo] [, transa] [, transb] [, alpha] [, beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gemmt routines compute a scalar-matrix-matrix product with general matrices and add the result to the upper or lower part of a scalar-matrix product. These routines are similar to the ?gemm routines, but they only access and update a triangular part of the square result matrix (see Application Notes below).

The operation is defined as

C := *alpha**op(*A*)*op(*B*) + *beta***C*,

where:

op(*X*) is one of op(*X*) = *X*, or op(*X*) = X^T, or op(*X*) = X^H,

alpha and *beta* are scalars,

A, *B* and *C* are matrices:

op(*A*) is an *n*-by-*k* matrix,

op(*B*) is a *k*-by-*n* matrix,

C is an *n*-by-*n* upper or lower triangular matrix.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is used. If *uplo* = 'U' or 'u', then the upper triangular part of the array *c* is used. If *uplo* = 'L' or 'l', then the lower triangular part of the array *c* is used.

transa

CHARACTER*1. Specifies the form of op(*A*) used in the matrix multiplication:

if *transa* = 'N' or 'n', then op(*A*) = *A*;

if *transa* = 'T' or 't', then op(*A*) = A^T;

if *transa* = 'C' or 'c', then op(*A*) = A^H.

transb

CHARACTER*1. Specifies the form of op(*B*) used in the matrix multiplication:

if *transb* = 'N' or 'n', then op(*B*) = *B*;

if *transb* = 'T' or 't', then op(*B*) = B^T;

if *transb* = 'C' or 'c', then op(*B*) = B^H.

n — INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

k — INTEGER. Specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*). The value of *k* must be at least zero.

alpha — REAL for sgemmt

DOUBLE PRECISION for dgemmt

COMPLEX for cgemmt

DOUBLE COMPLEX for zgemmt

Specifies the scalar *alpha*.

a — REAL for sgemmt

DOUBLE PRECISION for dgemmt

COMPLEX for cgemmt

DOUBLE COMPLEX for zgemmt

Array, size *lda* by *ka*, where *ka* is *k* when *transa* = 'N' or 'n', and is *n* otherwise. Before entry with *transa* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda — INTEGER.

Specifies the leading dimension of *a* as declared in the calling (sub)program.

When *transa* = 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).

b — REAL for sgemmt

DOUBLE PRECISION for dgemmt

COMPLEX for cgemmt

DOUBLE COMPLEX for zgemmt

Array, size *ldb* by *kb*, where *kb* is *n* when *transb* = 'N' or 'n', and is *k* otherwise. Before entry with *transb* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *n*-by-*k* part of the array *b* must contain the matrix *B*.

ldb — INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

When *transb* = 'N' or 'n', then *ldb* must be at least max(1, *k*), otherwise *ldb* must be at least max(1, *n*).

beta — REAL for sgemmt

DOUBLE PRECISION for dgemmt

`COMPLEX` for `cgemmt`

`DOUBLE COMPLEX` for `zgemmt`

Specifies the scalar *beta*. When `beta` is equal to zero, then `c` need not be set on input.

`c`

`REAL` for `sgemmt`

`DOUBLE PRECISION` for `dgemmt`

`COMPLEX` for `cgemmt`

`DOUBLE COMPLEX` for `zgemmt`

Array, size `ldc` by `n`.

Before entry with `uplo` = `'U'` or `'u'`, the leading `n`-by-`n` upper triangular part of the array `c` must contain the upper triangular part of the matrix *C* and the strictly lower triangular part of `c` is not referenced.

Before entry with `uplo` = `'L'` or `'l'`, the leading `n`-by-`n` lower triangular part of the array `c` must contain the lower triangular part of the matrix *C* and the strictly upper triangular part of `c` is not referenced.

When `beta` is equal to zero, `c` need not be set on input.

`ldc`

`INTEGER`. Specifies the leading dimension of `c` as declared in the calling (sub)program. The value of `ldc` must be at least max(1, `n`).

Output Parameters

`c`

When `uplo` = `'U'` or `'u'`, the upper triangular part of the array `c` is overwritten by the upper triangular part of the updated matrix.

When `uplo` = `'L'` or `'l'`, the lower triangular part of the array `c` is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran 95 Interface Conventions.

Specific details for the routine `gemmt` interface are the following:

`a`

Holds the matrix *A* of size (*ma*,*ka*) where

`ka` = `k` if `transa`=`'N'`,

`ka` = `n` otherwise,

`ma` = `n` if `transa`=`'N'`,

`ma` = `k` otherwise.

`b`

Holds the matrix *B* of size (*mb*,*kb*) where

`kb` = `n` if `transb` = `'N'`,

`kb` = `k` otherwise,

`mb` = `k` if `transb` = `'N'`,

	mb = *n* otherwise.
c	Holds the matrix *C* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
transa	Must be 'N', 'C', or 'T'. The default value is 'N'.
transb	Must be 'N', 'C', or 'T'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 0.

Application Notes

These routines only access and update the upper or lower triangular part of the result matrix. This can be useful when the result is known to be symmetric; for example, when computing a product of the form *C* := *alpha**B**S**B*^{T} + *beta**C* , where *S* and *C* are symmetric matrices and *B* is a general matrix. In this case, first compute *A* := *B***S* (which can be done using the corresponding ?symm routine), then compute *C* := *alpha**A**B*^{T} + *beta**C* using the ?gemmt routine.

?gemm3m

Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.

Syntax

```
call cgemm3m(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemm3m(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call gemm3m(a, b, c [,transa][,transb] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gemm3m routines perform a matrix-matrix operation with general complex matrices. These routines are similar to the ?gemm routines, but they use fewer matrix multiplication operations (see *Application Notes* below).

The operation is defined as

```
C := alpha*op(A)*op(B) + beta*C,
```

where:

op(*x*) is one of op(*x*) = *x*, or op(*x*) = *x*', or op(*x*) = conjg(*x*'),

alpha and *beta* are scalars,

A, *B* and *C* are matrices:

op(*A*) is an *m*-by-*k* matrix,

op(*B*) is a *k*-by-*n* matrix,

C is an *m*-by-*n* matrix.

Input Parameters

transa

CHARACTER*1. Specifies the form of op(*A*) used in the matrix multiplication:

if *transa* = 'N' or 'n', then op(*A*) = *A*;

if *transa* = 'T' or 't', then op(*A*) = *A*';

if *transa* = 'C' or 'c', then op(*A*) = conjg(*A*').

transb

CHARACTER*1. Specifies the form of op(*B*) used in the matrix multiplication:

if *transb* = 'N' or 'n', then op(*B*) = *B*;

if *transb* = 'T' or 't', then op(*B*) = *B*';

if *transb* = 'C' or 'c', then op(*B*) = conjg(*B*').

m

INTEGER. Specifies the number of rows of the matrix op(*A*) and of the matrix *C*. The value of m must be at least zero.

n

INTEGER. Specifies the number of columns of the matrix op(*B*) and the number of columns of the matrix *C*.

The value of *n* must be at least zero.

k

INTEGER. Specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*).

The value of *k* must be at least zero.

alpha

COMPLEX for cgemm3m

DOUBLE COMPLEX for zgemm3m

Specifies the scalar *alpha*.

a

COMPLEX for cgemm3m

DOUBLE COMPLEX for zgemm3m

Array, size *lda* by *ka*, where *ka* is *k* when *transa* = 'N' or 'n', and is *m* otherwise. Before entry with *transa* = 'N' or 'n', the leading *m*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*m* part of the array *a* must contain the matrix *A*.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

When *transa* = 'N' or 'n', then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1, *k*).

b

COMPLEX for cgemm3m

DOUBLE COMPLEX for zgemm3m

Array, size *ldb* by *kb*, where *kb* is *n* when *transa* = 'N' or 'n', and is *k* otherwise. Before entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *n*-by-*k* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

When *transa* = 'N' or 'n', then *ldb* must be at least max(1, *k*), otherwise *ldb* must be at least max(1, *n*).

beta

COMPLEX for cgemm3m

DOUBLE COMPLEX for zgemm3m

Specifies the scalar *beta*.

When *beta* is equal to zero, then *c* need not be set on input.

c

COMPLEX for cgemm3m

DOUBLE COMPLEX for zgemm3m

Array, size *ldc* by *n*. Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is equal to zero, in which case *c* need not be set on entry.

ldc

INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

The value of *ldc* must be at least max(1, *m*).

Output Parameters

c

Overwritten by the *m*-by-*n* matrix (*alpha**op(*A*)*op(*B*) + *beta***C*).

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine gemm3m interface are the following:

a

Holds the matrix *A* of size (*ma*,*ka*) where

ka = *k* if *transa*='N',

ka = *m* otherwise,

ma = *m* if *transa*='N',

ma = *k* otherwise.

b

Holds the matrix *B* of size (*mb*,*kb*) where

kb = *n* if *transb* = 'N',

kb = *k* otherwise,

mb = *k* if *transb* = 'N',

mb = *n* otherwise.

c	Holds the matrix *C* of size (*m*,*n*).
transa	Must be 'N', 'C', or 'T'. The default value is 'N'.
transb	Must be 'N', 'C', or 'T'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 1.

Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$fl(x \text{ op } y)=(x \text{ op } y)(1+\delta), |\delta|\le u, \text{ op}=\times,/, \quad fl(x\pm y)=x(1+\alpha)\pm y(1+\beta), \quad |\alpha|,|\beta|\le u$

then for an *n*-by-*n* matrix $\hat{C}=fl(C_1+iC_2)= fl((A_1+iA_2)(B_1+iB_2))=\hat{C}_1+i\hat{C}_2$, the following bounds are satisfied:

$\|\hat{C}_1-C_1\|\le 2(n+1)u\|A\|_\infty\|B\|_\infty+O(u^2)$,

$\|\hat{C}_2-C_2\|\le 4(n+4)u\|A\|_\infty\|B\|_\infty+O(u^2)$,

where $\|A\|_\infty=\max(\|A_1\|_\infty,\|A_2\|_\infty)$, and $\|B\|_\infty=\max(\|B_1\|_\infty,\|B_2\|_\infty)$.

Thus the corresponding matrix multiplications are stable.

?gemm_batch

Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.

Syntax

call sgemm_batch(*transa_array*, *transb_array*, *m_array*, *n_array*, *k_array*, *alpha_array*, *a_array*, *lda_array*, *b_array*, *ldb_array*, *beta_array*, *c_array*, *ldc_array*, *group_count*, *group_size*)

call dgemm_batch(*transa_array*, *transb_array*, *m_array*, *n_array*, *k_array*, *alpha_array*, *a_array*, *lda_array*, *b_array*, *ldb_array*, *beta_array*, *c_array*, *ldc_array*, *group_count*, *group_size*)

call cgemm_batch(*transa_array*, *transb_array*, *m_array*, *n_array*, *k_array*, *alpha_array*, *a_array*, *lda_array*, *b_array*, *ldb_array*, *beta_array*, *c_array*, *ldc_array*, *group_count*, *group_size*)

call zgemm_batch(*transa_array*, *transb_array*, *m_array*, *n_array*, *k_array*, *alpha_array*, *a_array*, *lda_array*, *b_array*, *ldb_array*, *beta_array*, *c_array*, *ldc_array*, *group_count*, *group_size*)

call sgemm_batch(*a_array*, *b_array*, *c_array*, *m_array*, *n_array*, *k_array*, *group_size* [,*transa_array*][,*transb_array*] [,*alpha_array*][,*beta_array*])

```
call dgemm_batch(a_array, b_array, c_array, m_array, n_array, k_array, group_size
[,transa_array][,transb_array] [,alpha_array][,beta_array])
```

```
call cgemm_batch(a_array, b_array, c_array, m_array, n_array, k_array, group_size
[,transa_array][,transb_array] [,alpha_array][,beta_array])
```

```
call zgemm_batch(a_array, b_array, c_array, m_array, n_array, k_array, group_size
[,transa_array][,transb_array] [,alpha_array][,beta_array])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gemm_batch routines perform a series of matrix-matrix operations with general matrices. They are similar to the ?gemm routine counterparts, but the ?gemm_batch routines perform matrix-matrix operations with groups of matrices, processing a number of groups at once. The groups contain matrices with the same parameters.

The operation is defined as

```
idx = 1
for i = 1..group_count
     alpha and beta in alpha_array(i) and beta_array(i)
     for j = 1..group_size(i)
          A, B, and C matrix in a_array(idx), b_array(idx), and c_array(idx)
          C := alpha*op(A)*op(B) + beta*C,
          idx = idx + 1
     end for
 end for
```

where:

op(*X*) is one of op(*X*) = *X*, or op(*X*) = X^T, or op(*X*) = X^H,

alpha and *beta* are scalar elements of *a_array* and *beta_array*,

A, *B* and *C* are matrices such that for *m*, *n*, and *k* which are elements of *m_array*, *n_array*, and *k_array*:

op(*A*) is an *m*-by-*k* matrix,

op(*B*) is a *k*-by-*n* matrix,

C is an *m*-by-*n* matrix.

A, *B*, and *C* represent matrices stored at addresses pointed to by *a_array*, *b_array*, and *c_array*, respectively. The number of entries in *a_array*, *b_array*, and *c_array* is *total_batch_count* = the sum of all of the *group_size* entries.

See also gemm for a detailed description of multiplication for general matrices and ?gemm3m_batch, BLAS-like extension routines for similar matrix-matrix operations.

NOTE
Error checking is not performed for oneMKL Windows* single dynamic libraries for the?gemm_batch routines.

Input Parameters

transa_array

CHARACTER*1. Array of size *group_count*. For the group *i*, $transa_i$ = *transa_array*(*i*) specifies the form of op(*A*) used in the matrix multiplication:

if $transa_i$ = 'N' or 'n', then op(*A*) = *A*;

if $transa_i$ = 'T' or 't', then op(*A*) = A^T;

if $transa_i$ = 'C' or 'c', then op(*A*) = A^H.

transb_array

CHARACTER*1. Array of size *group_count*. For the group *i*, $transb_i$ = *transb_array*(*i*) specifies the form of op(B_i) used in the matrix multiplication:

if $transb_i$ = 'N' or 'n', then op(*B*) = *B*;

if $transb_i$ = 'T' or 't', then op(*B*) = B^T;

if $transb_i$ = 'C' or 'c', then op(*B*) = B^H.

m_array

INTEGER. Array of size *group_count*. For the group *i*, m_i = *m_array*(*i*) specifies the number of rows of the matrix op(*A*) and of the matrix *C*.

The value of each element of *m_array* must be at least zero.

n_array

INTEGER. Array of size *group_count*. For the group *i*, n_i = *n_array*(*i*) specifies the number of columns of the matrix op(*B*) and the number of columns of the matrix *C*.

The value of each element of *n_array* must be at least zero.

k_array

INTEGER. Array of size *group_count*. For the group *i*, k_i = *k_array*(*i*) specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*).

The value of each element of *k_array* must be at least zero.

alpha_array

REAL for sgemm_batch

DOUBLE PRECISION for dgemm_batch

COMPLEX for cgemm_batch

DOUBLE COMPLEX for zgemm_batch

Array of size *group_count*. For the group *i*, *alpha_array*(*i*) specifies the scalar $alpha_i$.

a_array

INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA-32 architecture

Array, size *total_batch_count*, of pointers to arrays used to store *A* matrices.

lda_array

INTEGER. Array of size *group_count*. For the group *i*, lda_i = *lda_array*(*i*) specifies the leading dimension of the array storing matrix *A* as declared in the calling (sub)program.

When $transa_i$ = 'N' or 'n', then lda_i must be at least max(1, m_i), otherwise lda_i must be at least max(1, k_i).

b_array

`INTEGER*8` for Intel® 64 architecture

`INTEGER*4` for IA-32 architecture

Array, size *total_batch_count*, of pointers to arrays used to store *B* matrices.

ldb_array

`INTEGER`.

Array of size *group_count*. For the group *i*, ldb_i = *ldb_array*(*i*) specifies the leading dimension of the array storing matrix *B* as declared in the calling (sub)program.

When $transb_i$ = 'N' or 'n', then ldb_i must be at least max(1, k_i), otherwise ldb_i must be at least max(1, n_i).

beta_array

`REAL` for `sgemm_batch`

`DOUBLE PRECISION` for `dgemm_batch`

`COMPLEX` for `cgemm_batch`

`DOUBLE COMPLEX` for `zgemm_batch`

Array of size *group_count*. For the group *i*, *beta_array*(*i*) specifies the scalar $beta_i$.

When $beta_i$ is equal to zero, then *C* matrices in group *i* need not be set on input.

c_array

`INTEGER*8` for Intel® 64 architecture

`INTEGER*4` for IA-32 architecture

Array, size *total_batch_count*, of pointers to arrays used to store *C* matrices.

ldc_array

`INTEGER`.

Array of size *group_count*. For the group *i*, ldc_i = *ldc_array*(*i*) specifies the leading dimension of all arrays storing matrix *C* in group *i* as declared in the calling (sub)program.

ldc_i must be at least max(1, m_i).

group_count

`INTEGER`.

Specifies the number of groups. Must be at least 0.

group_size

`INTEGER`.

Array of size *group_count*. The element *group_size*(*i*) specifies the number of matrices in group *i*. Each element in *group_size* must be at least 0.

Output Parameters

c_array

Output buffer, overwritten by *total_batch_count* matrix multiply operations of the form alpha*op(A)*op(B) + beta*C.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `gemm_batch` interface are the following:

a_array	Holds pointers to arrays containing matrices *A* of size (*ma*,*ka*) where *ka* = *k* if *transa*='N', *ka* = *m* otherwise, *ma* = *m* if *transa*='N', *ma* = *k* otherwise.
b_array	Holds pointers to arrays containing matrices *B* of size (*mb*,*kb*) where *kb* = *n* if *transb_array* = 'N', *kb* = *k* otherwise, *mb* = *k* if *transb_array* = 'N', *mb* = *n* otherwise.
c_array	Holds pointers to arrays containing matrices *C* of size (*m*,*n*).
m_array	Array indicating number of rows of matrices op(*A*) and *C* for each group.
n_array	Array indicating number of columns of matrices op(*B*) and *C* for each group.
k_array	Array indicating number of columns of matrices op(*A*) and number of rows of matrices op(*B*) for each group.
group_size	Array indicating number of matrices for each group. Each element in *group_size* must be at least 0.
transa_array	Array with each element set to one of 'N', 'C', or 'T'. The default values are 'N'.
transb_array	Array with each element set to one of 'N', 'C', or 'T'. The default values are 'N'.
alpha_array	Array of *alpha* values; the default value is 1.
beta_array	Array of *beta* values; the default value is 0.

?gemm_batch_strided

Computes groups of matrix-matrix product with general matrices.

Syntax

```
call sgemm_batch_strided(transa, transb, m, n, k, alpha, a, lda, stridea, b, ldb,
strideb, beta, c, ldc, stridec, batch_size)
```

```
call dgemm_batch_strided(transa, transb, m, n, k, alpha, a, lda, stridea, b, ldb,
strideb, beta, c, ldc, stridec, batch_size)
```

```
call cgemm_batch_strided(transa, transb, m, n, k, alpha, a, lda, stridea, b, ldb,
strideb, beta, c, ldc, stridec, batch_size)
```

```
call zgemm_batch_strided(transa, transb, m, n, k, alpha, a, lda, stridea, b, ldb,
strideb, beta, c, ldc, stridec, batch_size)
```

Include Files

- mkl.fi

Description

The ?gemm_batch_strided routines perform a series of matrix-matrix operations with general matrices. They are similar to the ?gemm routine counterparts, but the ?gemm_batch_strided routines perform matrix-matrix operations with groups of matrices. The groups contain matrices with the same parameters.

All matrix *a* (respectively, *b* or *c*) have the same parameters (size, leading dimension, transpose operation, alpha, beta scaling) and are stored at constant *stridea* (respectively, *strideb* or *stridec*) from each other. The operation is defined as

```
For i = 0 … batch_size – 1
    Ai, Bi and Ci are matrices at offset i * stridea, i * strideb and i * stridec in a, b and c
    Ci = alpha * Ai * Bi +  beta * Ci
end for
```

Input Parameters

transa

CHARACTER*1.

Specifies op(A) the transposition operation applied to the matrices *A*.

if *transa* = 'N' or 'n' , then op(A) = A;

if *transa* = 'T' or 't' , then op(A) = A^T;

if *transa* = 'C' or 'c' , then op(A) = A^H.

transb

CHARACTER*1.

Specifies op(B) the transposition operation applied to the matrices *B*.

if *transb* = 'N' or 'n' , then op(B) = B;

if *transb* = 'T' or 't' , then op(B) = B^T;

if *transb* = 'C' or 'c' , then op(B) = B^H.

m

INTEGER. Number of rows of the op(A) and *C* matrices. Must be at least 0.

n

INTEGER. Number of columns of the op(B) and *C* matrices. Must be at least 0.

k

INTEGER. Number of columns of the op(A) matrix and number of rows of the op(B) matrix. Must be at least 0.

alpha

REAL for sgemm_batch_strided

DOUBLE PRECISION for dgemm_batch_strided

COMPLEX for cgemm_batch_strided

DOUBLE COMPLEX for zgemm_batch_strided

Specifies the scalar *alpha*.

a

REAL for sgemm_batch_strided

DOUBLE PRECISION for dgemm_batch_strided

COMPLEX for cgemm_batch_strided

DOUBLE COMPLEX for zgemm_batch_strided

Array of size at least *stridea***batch_size* holding the *a* matrices.

transa='N' or 'n'	*transa*='T' or 't' or 'C' or 'c'

lda

INTEGER. Specifies the leading dimension of the *a* matrices.

transa='N' or 'n'	*transa*='T' or 't' or 'C' or 'c'

stridea

INTEGER. Stride between two consecutive *a* matrices.

transa='N' or 'n'	*transa*='T' or 't' or 'C' or 'c'

b

REAL for sgemm_batch_strided

DOUBLE PRECISION for dgemm_batch_strided

COMPLEX for cgemm_batch_strided

DOUBLE COMPLEX for zgemm_batch_strided

Array of size at least *strideb***batch_size* holding the *b* matrices.

transb='N' or 'n'	*transb*='T' or 't' or 'C' or 'c'

ldb

INTEGER. Specifies the leading dimension of the *b* matrices.

transab='N' or 'n'	*transb*='T' or 't' or 'C' or 'c'

strideb

INTEGER. Stride between two consecutive *b* matrices.

transa='N' or 'n'	*transa*='T' or 't' or 'C' or 'c'

beta

REAL for sgemm_batch_strided

DOUBLE PRECISION for dgemm_batch_strided

COMPLEX for cgemm_batch_strided

DOUBLE COMPLEX for zgemm_batch_strided

Specifies the scalar *beta*.

c

REAL for sgemm_batch_strided

DOUBLE PRECISION for dgemm_batch_strided

COMPLEX for cgemm_batch_strided

DOUBLE COMPLEX for zgemm_batch_strided

Array of size at least *stridec***batch_size* holding the *c* matrices.

ldc INTEGER.

Specifies the leading dimension of the *c* matrices.

Must be at least max(1,*m*) .

stridec INTEGER.

Specifies the stride between two consecutive *c* matrices.

Must be at least ldc**n* .

batch_size INTEGER.

Number of gemm computations to perform and *a*, *b* and *c* matrices. Must be at least 0.

Output Parameters

c Array holding the *batch_size* updated *c* matrices.

?gemm3m_batch_strided

Computes groups of matrix-matrix product with general matrices.

Syntax

```
call cgemm3m_batch_strided(transa, transb, m, n, k, alpha, a, lda, stridea, b, ldb, strideb, beta, c, ldc, stridec, batch_size)
```

```
call zgemm3m_batch_strided(transa, transb, m, n, k, alpha, a, lda, stridea, b, ldb, strideb, beta, c, ldc, stridec, batch_size)
```

Include Files

- mkl.fi

Description

The ?gemm3m_batch_strided routines perform a series of matrix-matrix operations with general matrices. They are similar to the ?gemm routine counterparts, but the ?gemm3m_batch_strided routines perform matrix-matrix operations with groups of matrices. The groups contain matrices with the same parameters.

All matrix *a* (respectively, *b* or *c*) have the same parameters (size, leading dimension, transpose operation, alpha, beta scaling) and are stored at constant *stridea* (respectively, *strideb* or *stridec*) from each other. The operation is defined as

```
For i = 0 … batch_size – 1
    Ai, Bi and Ci are matrices at offset i * stridea, i * strideb and i * stridec in a, b and c
    Ci = alpha * Ai * Bi +  beta * Ci
end for
```

The ?gemm3m_batch_strided routines use fewer matrix multiplications than the ?gemm routines, as described in the *Application Notes* below.

Input Parameters

transa CHARACTER*1.

Specifies op(A) the transposition operation applied to the matrices *A*.

if *transa* = 'N' or 'n' , then op(A) = A;

if *transa* = 'T' or 't' , then op(A) = A^T;

if *transa* = 'C' or 'c' , then op(A) = A^H.

transb

CHARACTER*1.

Specifies op(B) the transposition operation applied to the matrices *B*.

if *transb* = 'N' or 'n' , then op(B) = B;

if *transb* = 'T' or 't' , then op(B) = B^T;

if *transb* = 'C' or 'c' , then op(B) = B^H.

m

INTEGER. Number of rows of the op(A) and *C* matrices. Must be at least 0.

n

INTEGER. Number of columns of the op(B) and *C* matrices. Must be at least 0.

k

INTEGER. Number of columns of the op(A) matrix and number of rows of the op(B) matrix. Must be at least 0.

alpha

COMPLEX for cgemm3m_batch_strided

DOUBLE COMPLEX for zgemm3m_batch_strided

Specifies the scalar *alpha*.

a

COMPLEX for cgemm3m_batch_strided

DOUBLE COMPLEX for zgemm3m_batch_strided

Array of size at least *stridea***batch_size* holding the *a* matrices.

transa='N' or 'n'	*transa*='T' or 't' or 'C' or 'c'

lda

INTEGER. Specifies the leading dimension of the *a* matrices.

transa='N' or 'n'	*transa*='T' or 't' or 'C' or 'c'

stridea

INTEGER. Stride between two consecutive *a* matrices.

transa='N' or 'n'	*transa*='T' or 't' or 'C' or 'c'

b

COMPLEX for cgemm3m_batch_strided

DOUBLE COMPLEX for zgemm3m_batch_strided

Array of size at least *strideb***batch_size* holding the *b* matrices.

transb='N' or 'n'	*transb*='T' or 't' or 'C' or 'c'

ldb

INTEGER. Specifies the leading dimension of the *b* matrices.

transab='N' or 'n'	*transb*='T' or 't' or 'C' or 'c'

strideb INTEGER. Stride between two consecutive *b* matrices.

transa='N' or 'n'	*transa*='T' or 't' or 'C' or 'c'

beta COMPLEX for cgemm3m_batch_strided

DOUBLE COMPLEX for zgemm3m_batch_strided

Specifies the scalar *beta*.

c COMPLEX for cgemm3m_batch_strided

DOUBLE COMPLEX for zgemm3m_batch_strided

Array of size at least *stridec***batch_size* holding the *c* matrices.

ldc INTEGER.

Specifies the leading dimension of the *c* matrices.

Must be at least max(1,*m*) .

stridec INTEGER.

Specifies the stride between two consecutive *c* matrices.

Must be at least ldc**n* .

batch_size INTEGER.

Number of gemm computations to perform and *a*, *b* and *c* matrices. Must be at least 0.

Output Parameters

c Array holding the *batch_size* updated *c* matrices.

Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$fl(x\ op\ y)=(x\ op\ y)(1+\delta),|\delta|\leq u,\ op=\times,/,\ fl(x\pm y)=x(1+\alpha)\pm y(1+\beta),\ |\alpha|,|\beta|\leq u$

then for an n-by-n matrix $\hat{C}=fl(C1+iC2)=fl((A1+iA2)(B1+iB2))=\hat{C}1+i\hat{C}2$, the following bounds are satisfied:

$$\|\hat{C}1-C1\|\leq 2(n+1)u\|A\|_\infty\|B\|_\infty+O(u2),$$
$$\|\hat{C}2-C2\|\leq 4(n+4)u\|A\|_\infty\|B\|_\infty+O(u2),$$

where $\|A\|_\infty=max(\|A1\|_\infty,\|A2\|_\infty),\ and\ \|B\|_\infty=max(\|B1\|_\infty,\|B2\|_\infty)$.

Thus the corresponding matrix multiplications are stable.

?gemm3m_batch

Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.

Syntax

`call cgemm3m_batch(transa_array, transb_array, m_array, n_array, k_array, alpha_array, a_array, lda_array, b_array, ldb_array, beta_array, c_array, ldc_array, group_count, group_size)`

`call zgemm3m_batch(transa_array, transb_array, m_array, n_array, k_array, alpha_array, a_array, lda_array, b_array, ldb_array, beta_array, c_array, ldc_array, group_count, group_size)`

`call cgemm3m_batch(a_array, b_array, c_array, m_array, n_array, k_array, group_size [,transa_array][,transb_array] [,alpha_array][,beta_array])`

`call zgemm3m_batch(a_array, b_array, c_array, m_array, n_array, k_array, group_size [,transa_array][,transb_array] [,alpha_array][,beta_array])`

Include Files

- `mkl.fi, blas.f90`

Description

The `?gemm3m_batch` routines perform a series of matrix-matrix operations with general matrices. They are similar to the `?gemm3m` routine counterparts, but the `?gemm3m_batch` routines perform matrix-matrix operations with groups of matrices, processing a number of groups at once. The groups contain matrices with the same parameters. The `?gemm3m_batch` routines use fewer matrix multiplications than the `?gemm_batch` routines, as described in the *Application Notes*.

The operation is defined as

```
idx = 1
for i = 1..group_count
     alpha and beta in alpha_array(i) and beta_array(i)
     for j = 1..group_size(i)
          A, B, and C matrix in a_array(idx), b_array(idx), and c_array(idx)
          C := alpha*op(A)*op(B) + beta*C,
          idx = idx + 1
     end for
 end for
```

where:

`op(X)` is one of `op(X) = X`, or `op(X) =` X^T, or `op(X) =` X^H,

alpha and *beta* are scalar elements of *alpha_array* and *beta_array*,

A, *B* and *C* are matrices such that for *m*, *n*, and *k* which are elements of *m_array*, *n_array*, and *k_array*:

`op(A)` is an *m*-by-*k* matrix,

`op(B)` is a *k*-by-*n* matrix,

C is an *m*-by-*n* matrix.

A, *B*, and *C* represent matrices stored at addresses pointed to by `a_array`, `b_array`, and `c_array`, respectively. The number of entries in `a_array`, `b_array`, and `c_array` is *total_batch_count* = the sum of all the `group_size` entries.

See also gemm for a detailed description of multiplication for general matrices and gemm_batch, BLAS-like extension routines for similar matrix-matrix operations.

NOTE
Error checking is not performed for Intel® oneAPI Math Kernel Library (oneMKL) Windows* single dynamic libraries for the `?gemm3m_batch` routines.

Input Parameters

`transa_array`

`CHARACTER*1`. Array of size `group_count`. For the group *i*, $transa_i$ = `transa_array(i)` specifies the form of `op(A)` used in the matrix multiplication:

if $transa_i$ = `'N'` or `'n'`, then `op(A)` = A;

if $transa_i$ = `'T'` or `'t'`, then `op(A)` = A^T;

if $transa_i$ = `'C'` or `'c'`, then `op(A)` = A^H.

`transb_array`

`CHARACTER*1`. Array of size `group_count`. For the group *i*, $transb_i$ = `transb_array(i)` specifies the form of `op(`B_i`)` used in the matrix multiplication:

if $transb_i$ = `'N'` or `'n'`, then `op(B)` = B;

if $transb_i$ = `'T'` or `'t'`, then `op(B)` = B^T;

if $transb_i$ = `'C'` or `'c'`, then `op(B)` = B^H.

`m_array`

`INTEGER`. Array of size `group_count`. For the group *i*, m_i = `m_array(i)` specifies the number of rows of the matrix `op(A)` and of the matrix *C*.

The value of each element of `m_array` must be at least zero.

`n_array`

`INTEGER`. Array of size `group_count`. For the group *i*, n_i = `n_array(i)` specifies the number of columns of the matrix `op(B)` and the number of columns of the matrix *C*.

The value of each element of `n_array` must be at least zero.

`k_array`

`INTEGER`. Array of size `group_count`. For the group *i*, k_i = `k_array(i)` specifies the number of columns of the matrix `op(A)` and the number of rows of the matrix `op(B)`.

The value of each element of `k_array` must be at least zero.

`alpha_array`

`COMPLEX` for `cgemm3m_batch`

`DOUBLE COMPLEX` for `zgemm3m_batch`

Array of size `group_count`. For the group *i*, `alpha_array(i)` specifies the scalar $alpha_i$.

`a_array`

`INTEGER*8` for Intel® 64 architecture

`INTEGER*4` for IA-32 architecture

Array, size *total_batch_count*, of pointers to arrays used to store *A* matrices.

lda_array

INTEGER. Array of size *group_count*. For the group *i*, lda_i = *lda_array*(*i*) specifies the leading dimension of the array storing matrix *A* as declared in the calling (sub)program.

When $transa_i$ = 'N' or 'n', then lda_i must be at least max(1, m_i), otherwise lda_i must be at least max(1, k_i).

b_array

INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA-32 architecture

Array, size *total_batch_count*, of pointers to arrays used to store *B* matrices.

ldb_array

INTEGER.

Array of size *group_count*. For the group *i*, ldb_i = *ldb_array*(*i*) specifies the leading dimension of the array storing matrix *B* as declared in the calling (sub)program.

When $transb_i$ = 'N' or 'n', then ldb_i must be at least max(1, k_i), otherwise ldb_i must be at least max(1, n_i).

beta_array

COMPLEX for cgemm3m_batch

DOUBLE COMPLEX for zgemm3m_batch

For the group *i*, *beta_array*(*i*) specifies the scalar $beta_i$.

When $beta_i$ is equal to zero, then *C* matrices in group *i* need not be set on input.

c_array

INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA-32 architecture

Array, size *total_batch_count*, of pointers to arrays used to store *C* matrices.

ldc_array

INTEGER.

Array of size *group_count*. For the group *i*, ldc_i = *ldc_array*(*i*) specifies the leading dimension of all arrays storing matrix *C* in group *i* as declared in the calling (sub)program.

ldc_i must be at least max(1, m_i).

group_count

INTEGER.

Specifies the number of groups. Must be at least 0.

group_size

INTEGER.

Array of size *group_count*. The element *group_size*(*i*) specifies the number of matrices in group *i*. Each element in *group_size* must be at least 0.

Output Parameters

c_array — Overwritten by the m_i-by-n_i matrix `(alpha_i*op(A)*op(B) + beta_i*C)` for group *i*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `gemm3m_batch` interface are the following:

a_array — Holds pointers to arrays containing matrices *A* of size (*ma*,*ka*) where

ka = *k* if *transa*='N',

ka = *m* otherwise,

ma = *m* if *transa*='N',

ma = *k* otherwise.

b_array — Holds pointers to arrays containing matrices *B* of size (*mb*,*kb*) where

kb = *n* if *transb_array* = 'N',

kb = *k* otherwise,

mb = *k* if *transb_array* = 'N',

mb = *n* otherwise.

c_array — Holds pointers to arrays containing matrices *C* of size (*m*,*n*).

m_array — Array indicating number of rows of matrices `op(A)` and *C* for each group.

n_array — Array indicating number of columns of matrices `op(B)` and *C* for each group.

k_array — Array indicating number of columns of matrices `op(A)` and number of rows of matrices `op(B)` for each group.

group_size — Array indicating number of matrices for each group. Each element in *group_size* must be at least 0.

transa_array — Array with each element set to one of 'N', 'C', or 'T'.

The default values are 'N'.

transb_array — Array with each element set to one of 'N', 'C', or 'T'.

The default values are 'N'.

alpha_array — Array of *alpha* values; the default value is 1.

beta_array — Array of *beta* values; the default value is 0.

Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$fl(x \text{ op } y)=(x \text{ op } y)(1+\delta), |\delta| \le u, \text{ op}=\times,/, \quad fl(x \pm y)=x(1+\alpha) \pm y(1+\beta), \quad |\alpha|, |\beta| \le u$

then for an *n*-by-*n* matrix $\hat{C}=fl(C_1+iC_2)= fl((A_1+iA_2)(B_1+iB_2))=\hat{C}_1+i\hat{C}_2$, the following bounds are satisfied:

$\|\hat{C}_1-C_1\| \le 2(n+1)u\|A\|_\infty\|B\|_\infty+O(u^2)$,

$\|\hat{C}_2-C_2\| \le 4(n+4)u\|A\|_\infty\|B\|_\infty+O(u^2)$,

where $\|A\|_\infty=\max(\|A_1\|_\infty, \|A_2\|_\infty)$, and $\|B\|_\infty=\max(\|B_1\|_\infty, \|B_2\|_\infty)$.

Thus the corresponding matrix multiplications are stable.

?trsm_batch

Solves a triangular matrix equation for a group of matrices.

Syntax

```
call strsm_batch(side_array, uplo_array, transa_array, diag_array, m_array, n_array,
alpha_array, a_array, lda_array, b_array, ldb_array, group_count, group_size)
```

```
call dtrsm_batch(side_array, uplo_array, transa_array, diag_array, m_array, n_array,
alpha_array, a_array, lda_array, b_array, ldb_array, group_count, group_size)
```

```
call ctrsm_batch(side_array, uplo_array, transa_array, diag_array, m_array, n_array,
alpha_array, a_array, lda_array, b_array, ldb_array, group_count, group_size)
```

```
call ztrsm_batch(side_array, uplo_array, transa_array, diag_array, m_array, n_array,
alpha_array, a_array, lda_array, b_array, ldb_array, group_count, group_size)
```

Include Files

- mkl.fi, blas.f90

Description

The ?trsm_batch routines solve a series of matrix equations. They are similar to the ?trsm routines except that they operate on groups of matrices which have the same parameters. The ?trsm_batch routines process a number of groups at once.

```
idx = 1
for i = 1..group_count
    alpha in alpha_array(i)
    for j = 1..group_size(i)
        A and B matrix in a_array(idx) and b_array(idx)
        Solve op(A)*X = alpha*B
          or
        Solve X*op(A) = alpha*B
```

```
        idx = idx + 1
    end for
end for
```

where:

alpha is a scalar element of `alpha_array`,

X and *B* are *m*-by-*n* matrices for *m* and *n* which are elements of `m_array` and `n_array`, respectively,

A is a unit, or non-unit, upper or lower triangular matrix,

and `op(A)` is one of `op(A) = A`, or `op(A) = A`^{T}, or `op(A) = conjg(A`^{T}`)`.

A and *B* represent matrices stored at addresses pointed to by `a_array` and `b_array`, respectively. There are *total_batch_count* entries in each of `a_array` and `b_array`, where *total_batch_count* is the sum of all the `group_size` entries.

Input Parameters

`side_array`

`CHARACTER*1`. Array of size *group_count*. For group *i*, 1 ≤*i*≤*group_count*, $side_i$ = `side_array(i)` specifies whether `op(A)` appears on the left or right of *X* in the equation:

if $side_i$ = `'L'` or `'l'`, then `op(A)*X = alpha*B`;

if $side_i$ = `'R'` or `'r'`, then `X*op(A) = alpha*B`.

`uplo_array`

`CHARACTER*1`. Array of size *group_count*. For group *i*, 1 ≤*i*≤*group_count*, $uplo_i$ = `uplo_array(i)` specifies whether the matrix *A* is upper or lower triangular:

$uplo_i$ = `'U'` or `'u'`

if $uplo_i$ = `'L'` or `'l'`, then the matrix is low triangular.

`transa_array`

`CHARACTER*1`. Array of size *group_count*. For group *i*, 1 ≤*i*≤*group_count*, $transa_i$ = `transa_array(i)` specifies the form of `op(A)` used in the matrix multiplication:

if $transa_i$= `'N'` or `'n'`, then `op(A) = A`;

if $transa_i$= `'T'` or `'t'`;

if $transa_i$= `'C'` or `'c'`, then `op(A) = conjg(A')`.

`diag_array`

`CHARACTER*1`. Array of size *group_count*. For group *i*, 1 ≤*i*≤*group_count*, $diag_i$ = `diag_array(i)` specifies whether the matrix *A* is unit triangular:

if $diag_i$ = `'U'` or `'u'` then the matrix is unit triangular;

if $diag_i$ = `'N'` or `'n'`, then the matrix is not unit triangular.

`m_array`

`INTEGER`. Array of size *group_count*. For group *i*, 1 ≤*i*≤*group_count*, m_i = `m_array(i)` specifies the number of rows of *B*. The value of m_i must be at least zero.

`n_array`

`INTEGER`. Array of size *group_count*. For group *i*, 1 ≤*i*≤*group_count*, n_i = `n_array(i)` specifies the number of columns of *B*. The value of n_i must be at least zero.

`alpha_array`

`REAL` for `strsm_batch`

`DOUBLE PRECISION` for `dtrsm_batch`

	`COMPLEX` for `ctrsm_batch`
	`DOUBLE COMPLEX` for `ztrsm_batch`
	Array of size *group_count*. For group *i*, 1 ≤*i*≤*group_count*, `alpha_array(i)` specifies the scalar $alpha_i$.

a_array	`INTEGER*8` for Intel® 64 architecture `INTEGER*4` for IA-32 architecture Array, size *total_batch_count*, of pointers to arrays used to store *A* matrices. For group *i*, 1 ≤*i*≤*group_count*, *k* is m_i when $side_i$ `= 'L'` or `'l'` and is n_i when *side* `= 'R'` or `'r'` and *a* is any of the `group_size(i)` arrays starting with `a_array(group_size(1) + group_size(2) + ... + group_size(i - 1) + 1)`: Before entry with $uplo_i$ `= 'U'` or `'u'`, the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with $uplo_i$ `= 'L'` or `'l'` lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced. When $diag_i$ `= 'U'` or `'u'`, the diagonal elements of *a* are not referenced either, but are assumed to be unity.
lda_array	`INTEGER`. Array of size *group_count*. For group *i*, 1 ≤*i*≤*group_count*, lda_i = `lda_array(i)` specifies the leading dimension of *a* as declared in the calling (sub)program. When $side_i$ `= 'L'` or `'l'`, then lda_i must be at least `max(1, `m_i`)`, when $side_i$ `= 'R'` or `'r'`, then lda_i must be at least `max(1, `n_i`)`.

b_array	`REAL` for `strsm_batch`
	`DOUBLE PRECISION` for `dtrsm_batch`
	`COMPLEX` for `ctrsm_batch`
	`DOUBLE COMPLEX` for `ztrsm_batch`
	Array, size *total_batch_count*, of pointers to arrays used to store *B* matrices.
	For group *i*, 1 ≤*i*≤*group_count*, *b* is any of the `group_size(i)` arrays starting with `b_array(group_size(1) + group_size(2) + ... + group_size(i - 1) + 1)`:
	Before entry, the leading m_i-by-n_i part of array *b* must contain the matrix *B*.

ldb_array	`INTEGER`. Array of size *group_count*. Specifies the leading dimension of *b* as declared in the calling (sub)program. *ldb* must be at least `max(1, `m`)`. `INTEGER`. Array of size *group_count*. For group *i*, 1 ≤*i*≤*group_count*, ldb_i = `ldb_array(i)` specifies the leading dimension of *b* as declared in the calling (sub)program. ldb_i must be at least `max(1, `m_i`)`.
group_count	`INTEGER`.

Specifies the number of groups. Must be at least 0.

group_size — INTEGER.

Array of size *group_count*. The element *group_size*(*i*) specifies the number of matrices in group *i*. Each element in *group_size* must be at least 0.

Output Parameters

b_array — Overwritten by the solution matrix *X*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see BLAS 95 Interface Conventions.

Specific details for the routine `trsm_batch` interface are the following:

a_array — Holds pointers to matrices *A* of size (*ma*,*ma*) where *ma* = m_i if *side* = 'L', *ma* = n_i otherwise.

b_array — Holds pointers to matrices *B* of size (m_i,n_i).

m_array — Array indicating the number of rows of matrix *C* for each group.

n_array — Array indicating the number of columns of matrix *B* for each group.

group_size — Array indicating the number of matrices for each group. Each element in *group_size* must be at least zero.

side_array — Array with each element set to either 'L' or 'R'. The default value is 'L'.

uplo_array — Array with each element set to either 'U' or 'L'. The default value is 'U'.

transa_array — Array with each element set to one of 'N', 'C', or 'T'.

The default value is 'N'.

diag_array — Array with each element set to either 'N' or 'U'. The default value is 'N'.

alpha_array — Array of *alpha*values. The default value is 1.

?trsm_batch_strided

Solves groups of triangular matrix equations.

Syntax

```
call strsm_batch_strided(side, uplo, transa, diag, m, n, alpha, a, lda, stridea, b, ldb, strideb, batch_size)
call dtrsm_batch_strided(side, uplo, transa, diag, m, n, alpha, a, lda, stridea, b, ldb, strideb, batch_size)
call ctrsm_batch_strided(side, uplo, transa, diag, m, n, alpha, a, lda, stridea, b, ldb, strideb, batch_size)
call ztrsm_batch_strided(side, uplo, transa, diag, m, n, alpha, a, lda, stridea, b, ldb, strideb, batch_size)
```

Include Files

- mkl.fi, blas.f90

Description

The ?trsm_batch_strided routines solve a series of triangular matrix equations. They are similar to the ?trsm routine counterparts, but the ?trsm_batch_strided routines solve triangular matrix equations with groups of matrices. All matrix *a* have the same parameters (*size*, *leading dimension*, *side*, *uplo*, *diag*, *transpose operation*) and are stored at constant *stridea* from each other. Similarly, all matrix *b* have the same parameters (*size*, *leading dimension*, *alpha scaling*) and are stored at constant *strideb* from each other.

The operation is defined as

```
For i = 0 … batch_size – 1
    Ai, and Bi are matrices at offset i * stridea and i * strideb in a and b
    Solve op(Ai)*Xi = alpha * Bi
        Or
    Solve Xi*op(Ai) = alpha * Bi
end for
```

Input Parameters

side

CHARACTER*1.

Specifies whether op(*A*) appears on the left or right of *X* in the equation.

if *side* = '*L*' or '*l*', then op(*A*)***X* = *alpha***B*;

if *side* = '*R*' or '*r*', then *X**op(*A*) = *alpha***B*.

uplo

CHARACTER*1.

Specifies whether the matrices *A* are upper or lower triangular.

if *uplo* = '*U*' or '*u*', then *A* are upper triangular;

if *uplo* = '*L*' or '*l*', then *A* are lower triangular.

transa

CHARACTER*1.

Specifies op(*A*) the transposition operation applied to the matrices *A*.

if *transa* = '*N*' or '*n*', then op(*A*) = *A*;

if *transa* = '*T*' or '*t*', then op(*A*) = A^T;

if *transa* = '*C*' or '*c*', then op(*A*) = A^H;

diag

CHARACTER*1.

Specifies whether the matrices *A* are unit triangular.

if *diag* = '*U*' or '*u*', then *A* are unit triangular;

if *diag* = '*N*' or '*n*', then *A* are non-unit triangular.

m

INTEGER.

Number of rows of *B* matrices. Must be at least 0

n

INTEGER.

Number of columns of *B* matrices. Must be at least 0

alpha

REAL for strsm_batch_strided

DOUBLE PRECISION for dtrsm_batch_strided

COMPLEX for ctrsm_batch_strided

DOUBLE COMPLEX for ztrsm_batch_strided

Specifies the scalar *alpha*.

a

REAL for strsm_batch_strided

DOUBLE PRECISION for dtrsm_batch_strided

COMPLEX for ctrsm_batch_strided

DOUBLE COMPLEX for ztrsm_batch_strided

Array of size at least *stridea***batch_size* holding the *A* matrices. Each *A* matrix is stored at constant *stridea* from each other.

Each *A* matrix has size *lda* k*, where *k* is *m* when *side* = '*L*' or '*l*' and is *n* when *side* = '*R*' or '*r*' .

Before entry with *uplo* = '*U*' or '*u*', the leading *k*-by-*k* upper triangular part of the array *A* must contain the upper triangular matrix and the strictly lower triangular part of *A* is not referenced.

Before entry with *uplo* = '*L*' or '*l*' lower triangular part of the array *A* must contain the lower triangular matrix and the strictly upper triangular part of *A* is not referenced.

When diag = '*U*' or '*u*' , the diagonal elements of *A* are not referenced either, but are assumed to be unity.

lda

INTEGER.

Specifies the leading dimension of the *A* matrices. When *side* = '*L*' or '*l*', then *lda* must be at least max(1, m), when side = *side* = '*R*' or '*r*', then *lda* must be at least max(1, n).

stridea

INTEGER.

Stride between two consecutive *A* matrices.

When *side* = '*L*' or '*l*', then *stridea* must be at least *lda*m*.

When side = *side* = '*R*' or '*r*', then *stridea* must be at least *lda*n*.

b

REAL for strsm_batch_strided

DOUBLE PRECISION for dtrsm_batch_strided

COMPLEX for ctrsm_batch_strided

DOUBLE COMPLEX for ztrsm_batch_strided

Array of size at least *strideb***batch_size* holding the *B* matrices. Each *B* matrix is stored at constant *strideb* from each other.

Each *B* matrix has size *ldb* n*. Before entry, the leading *m*-by-*n* part of the array *B* must contain the matrix *B*.

ldb

INTEGER.

Specifies the leading dimension of the *B* matrices.

ldb must be at least max(1,*m*).

strideb INTEGER.

Stride between two consecutive *B* matrices.

strideb must be at least `(ldb*n)`.

batch_size INTEGER.

Number of `trsm` computations to perform. Must be at least 0.

Output Parameters

b Overwritten by the solution batch_size *X* matrices.

mkl_?imatcopy

Performs scaling and in-place transposition/copying of matrices.

Syntax

```
call mkl_simatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)
call mkl_dimatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)
call mkl_cimatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)
call mkl_zimatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)
```

Include Files

- `mkl.fi`

Description

The `mkl_?imatcopy` routine performs scaling and in-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

AB := *alpha*op(*AB*).

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

NOTE
Different arrays must not overlap.

Input Parameters

ordering CHARACTER*1. Ordering of the matrix storage.

If *ordering* = `'R'` or `'r'`, the ordering is row-major.

If *ordering* = `'C'` or `'c'`, the ordering is column-major.

trans CHARACTER*1. Parameter that specifies the operation type.

If *trans* = 'N' or 'n', op(*AB*)=*AB* and the matrix *AB* is assumed unchanged on input.

If *trans* = 'T' or 't', it is assumed that *AB* should be transposed.

If *trans* = 'C' or 'c', it is assumed that *AB* should be conjugate transposed.

If *trans* = 'R' or 'r', it is assumed that *AB* should be only conjugated.

If the data is real, then *trans* = 'R' is the same as *trans* = 'N', and *trans* = 'C' is the same as *trans* = 'T'.

rows INTEGER. The number of rows in matrix *AB* before the transpose operation.

cols INTEGER. The number of columns in matrix *AB* before the transpose operation.

ab REAL for mkl_simatcopy.

DOUBLE PRECISION for mkl_dimatcopy.

COMPLEX for mkl_cimatcopy.

DOUBLE COMPLEX for mkl_zimatcopy.

Array, size *ab*(*lda*,*).

alpha REAL for mkl_simatcopy.

DOUBLE PRECISION for mkl_dimatcopy.

COMPLEX for mkl_cimatcopy.

DOUBLE COMPLEX for mkl_zimatcopy.

This parameter scales the input matrix by *alpha*.

lda INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix; measured in the number of elements.

This parameter must be at least *rows* if *ordering* = 'C' or 'c', and max(1,*cols*) otherwise.

ldb INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix; measured in the number of elements.

To determine the minimum value of *ldb* on output, consider the following guideline:

If *ordering* = 'C' or 'c', then

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least max(1,*cols*)
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least max(1,*rows*)

If *ordering* = 'R' or 'r', then

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least max(1,*rows*)

- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least max(1,*cols*)

Output Parameters

ab

REAL for mkl_simatcopy.

DOUBLE PRECISION for mkl_dimatcopy.

COMPLEX for mkl_cimatcopy.

DOUBLE COMPLEX for mkl_zimatcopy.

Array, size *ab*(*ldb*,*).

Contains the matrix *AB*.

Application Notes

For threading to be active in mkl_?imatcopy, the pointer *AB* must be aligned on the 64-byte boundary. This requirement can be met by allocating *AB* with mkl_malloc.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_simatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, src_ld, dst_ld
   REAL ab(*), alpha*
```

```
SUBROUTINE mkl_dimatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, src_ld, dst_ld
   DOUBLE PRECISION ab(*), alpha*
```

```
SUBROUTINE mkl_cimatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, src_ld, dst_ld
   COMPLEX ab(*), alpha*
```

```
SUBROUTINE mkl_zimatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, src_ld, dst_ld
   DOUBLE COMPLEX ab(*), alpha*
```

mkl_?imatcopy_batch

Computes a group of in-place scaled matrix copy or transposition operations on general matrices.

Syntax

call mkl_simatcopy_batch(*layout*, *trans_arrau*, *rows_arrau*, *cols_array*, *alpha_array*, *AB_array*, *lda_array*, *ldb_array*, *group_size*, *group_count*)

call mkl_dimatcopy_batch(*layout*, *trans_arrau*, *rows_arrau*, *cols_array*, *alpha_array*, *AB_array*, *lda_array*, *ldb_array*, *group_size*, *group_count*)

```
call mkl_cimatcopy_batch(layout, trans_arrau, rows_arrau, cols_array, alpha_array,
AB_array, lda_array, ldb_array, group_size, group_count)
```

```
call mkl_zimatcopy_batch(layout, trans_arrau, rows_arrau, cols_array, alpha_array,
AB_array, lda_array, ldb_array, group_size, group_count)
```

Description

The `mkl_?imatcopy_batch` routine performs a series of in-place scaled matrix copies or transpositions. They are similar to the `mkl_?imatcopy` routine counterparts, but the `mkl_?imatcopy_batch` routine performs matrix operations with groups of matrices. Each group has the same parameters (matrix size, leading dimension, and scaling parameter), but a single call to `mkl_?imatcopy_batch` operates on multiple groups, and each group can have different parameters, unlike the related `mkl_?imatcopy_batch_strided` routines.

The operation is defined as

```
idx = 0
for i = 0..group_count - 1
     m in rows_array[i], n in cols_array[i], and alpha in alpha_array[i]
     for j = 0..group_size[i] - 1
          AB matrices in AB_array[idx]
          AB := alpha*op(AB)
          idx = idx + 1
     end for
end for
```

Where `op(X)` is one of `op(X)=X`, `op(X)=X'`, `op(X)=conjg(X')`, or `op(X)=conjg(X)`. On entry, *AB* is a *m*-by-*n* matrix such that *m* and *n* are elements of *rows_array* and *cols_array*.

AB represents a matrix stored at addresses pointed to by *AB_array*. The number of entries in *AB_array* is *total_batch_count* = the sum of all of the *group_size* entries.

Input Parameters

layout

`CHARACTER*1`.

Specifies whether two-dimensional array storage is row-major (R) or column-major (C).

trans_array

`CHARACTER*1`.

Array of size *group_count*. For the group *i*, *trans* = *trans_array*[*i*] specifies the form of `op(AB)`, the transposition operation applied to the *AB* matrix:

If *trans* = `'N'` or `'n'`, `op(AB)=AB`.

If *trans* = `'T'` or `'t'`, `op(AB)=AB'`

If *trans* = `'C'` or `'c'`, `op(AB)=conjg(AB')`

If *trans* = `'R'` or `'r'`, `op(AB)=conjg(AB)`

rows_array

`INTEGER`. Array of size *group_count*. Specifies the number of rows of the input matrix *AB*. The value of each element must be at least zero.

cols_array

`INTEGER`. Array of size *group_count*. Specifies the number of columns of the input matrix *AB*. The value of each element must be at least zero.

alpha_array

`REAL` for `mkl_simatcopy_batch`.

`DOUBLE PRECISION` for `mkl_dimatcopy_batch`.

	`COMPLEX` for `mkl_cimatcopy_batch`. `DOUBLE COMPLEX` for `mkl_zimatcopy_batch`. Array of size *group_count*. Specifies the scalar *alpha*.
AB_array	`INTEGER*8` for Intel® 64 architecture. `INTEGER*4` for IA-32 architecture. Array of size *total_batch_count*, holding pointers to arrays used to store *AB* matrices.
lda_array	`INTEGER`. Array of size *group_count*. The leading dimension of the matrix input `AB`. It must be positive and at least *m* if column major layout is used or at least *n* if row major layout is used.
ldb_array	`INTEGER`. Array of size *group_count*. The leading dimension of the matrix input `AB`. It must be positive and at least *m* if column major layout is used and `op(AB) = AB` or `conjg(AB)` *n* if row major layout is used and `op(AB) = AB'` or `conjg(AB')` *n* otherwise
group_count	`INTEGER`. Specifies the number of groups. Must be at least 0
group_size	`INTEGER`. Array of size *group_count*. The element *group_size*[*i*] specifies the number of matrices in group *i*. Each element in *group_size* must be at least 0.

Output Parameters

AB_array	`INTEGER*8` for Intel® 64 architecture. `INTEGER*4` for IA-32 architecture. Output array of size *total_batch_count*, holding pointers to arrays used to store the updated *AB* matrices.

mkl_?imatcopy_batch_strided

Computes a group of in-place scaled matrix copy or transposition using general matrices.

Syntax

```
call mkl_simatcopy_batch_strided(layout, trans, row, col, alpha, ab, lda, ldb, stride, batch_size)
call mkl_dimatcopy_batch_strided(layout, trans, row, col, alpha, ab, lda, ldb, stride, batch_size)
call mkl_cimatcopy_batch_strided(layout, trans, row, col, alpha, ab, lda, ldb, stride, batch_size)
call mkl_zimatcopy_batch_strided(layout, trans, row, col, alpha, ab, lda, ldb, stride, batch_size)
```

Description

The mkl_?imatcopy_batch_strided routine performs a series of scaled matrix copy or transposition. They are similar to the mkl_?imatcopy routine counterparts, but the mkl_?imatcopy_batch_strided routine performs matrix operations with a group of matrices.

All matrices ab have the same parameters (size, transposition operation…) and are stored at constant stride from each other. The operation is defined as

```
for i = 0 … batch_size – 1
    AB is a matrix at offset i * stride in ab
    AB = alpha * op(AB)
end for
```

Input Parameters

layout

CHARACTER*1.

Specifies whether two-dimensional array storage is row-major or column-major

trans

CHARACTER*1. Specifies op(AB), the transposition operation applied to the AB matrices.

If *trans* = 'N' or 'n', op(*AB*)=*AB*.

If *trans* = 'T' or 't', op(*AB*)=*AB'*

If *trans* = 'C' or 'c', op(*AB*)=*conjg(AB')*

If *trans* = 'R' or 'r', op(*AB*)=*conjg(AB)*

row

INTEGER. Specifies the number of rows of the matrices AB. The value of *row* must be at least zero.

col

INTEGER. Specifies the number of columns of the matrices AB. The value of *col* must be at least zero.

alpha

REAL for mkl_simatcopy_batch_strided.

DOUBLE PRECISION for mkl_dimatcopy_batch_strided.

COMPLEX for mkl_cimatcopy_batch_strided.

DOUBLE COMPLEX for mkl_zimatcopy_batch_strided.

Specifies the scalar *alpha*.

ab

REAL for mkl_simatcopy_batch_strided.

DOUBLE PRECISION for mkl_dimatcopy_batch_strided.

COMPLEX for mkl_cimatcopy_batch_strided.

DOUBLE COMPLEX for mkl_zimatcopy_batch_strided.

Array holding all the input matrix AB. Must be of size at least batch_size * stride.

lda

INTEGER. The leading dimension of the matrix input AB. It must be positive and at least *row* if column major layout is used or at least *col* if row major layout is used.

ldb — INTEGER. The leading dimension of the matrix input AB. It must be positive and at least

row if column major layout is used and op(AB) = AB or conjg(AB)

row if row major layout is used and op(AB) = AB' or conjg(AB')

col otherwise

stride — INTEGER. Stride between two consecutive AB matrices, must be at least max(ldb,lda)*max(ka, kb) where

- ka is *row* if column major layout is used or *col* if row major layout is used
- kb is *col* if column major layout is used and op(AB) = AB or conjg(AB) or row major layout is used and op(AB) = AB' or conjg(AB'); kb is *row* otherwise.

batch_size — INTEGER. Number of imatcopy computations to perform and AB matrices. Must be at least 0.

Output Parameters

ab — Array holding the *batch_size* updated matrices AB.

mkl_?omatadd_batch_strided

Computes a group of out-of-place scaled matrix additions using general matrices.

Syntax

```
call mkl_somatadd_batch_strided(ordering, transa, transb, rows, cols, alpha, A, lda,
stridea, beta, B, ldb, strideb, C, ldc, stridec, batch_size);

call mkl_domatadd_batch_strided(ordering, transa, transb, rows, cols, alpha, A, lda,
stridea, beta, B, ldb, strideb, C, ldc, stridec, batch_size);

call mkl_comatadd_batch_strided(ordering, transa, transb, rows, cols, alpha, A, lda,
stridea, beta, B, ldb, strideb, C, ldc, stridec, batch_size);

call mkl_zomatadd_batch_strided(ordering, transa, transb, rows, cols, alpha, A, lda,
stridea, beta, B, ldb, strideb, C, ldc, stridec, batch_size);
```

Description

The mkl_omatadd_batch_strided routines perform a series of scaled matrix additions. They are similar to the mkl_omatadd routines, but the mkl_omatadd_batch_strided routines perform matrix operations with a group of matrices.

The matrices A, B, and C are stored at a constant stride from each other in memory, given by the parameters stridea, strideb, and stridec. The operation is defined as:

```
for i = 0 … batch_size – 1
    A is a matrix at offset i * stridea in the array a
    B is a matrix at offset i * strideb in the array b
    C is a matrix at offset i * stridec in the array c
    C = alpha * op(A) + beta * op(B)
end for
```

where:

- op(X) is one of op(X) = X, op(X) = X', op(X) = conjg(X) or op(X) = conjg(X').
- alpha and beta are scalars.
- A, B, and C are matrices.

The input arrays a and b contain all the input matrices, and the single output array c contains all the output matrices. The locations of the individual matrices within the array are given by stride lengths, while the number of matrices is given by the batch_size parameter.

Input Parameters

layout	**CHARACTER*** Specifies whether two-dimensional array storage is row-major or column-major.
transa	**CHARACTER*** Specifies op(A), the transposition operation applied to the matrices A. 'N' or 'n' indicates no operation, 'T' or 't' is transposition, 'R' or 'r' is complex conjugation wtihout tranpsosition, and 'C' or 'c' is conjugate transposition.
transb	**CHARACTER*** Specifies op(B), the transposition operation applied to the matrices B.
rows	**INTEGER** Number of rows for the result matrix C. Must be at least zero.
cols	**INTEGER** Number of columns for the result matrix C. Must be at least zero.
alpha	**REAL** for mkl_somatadd_batch_strided, *DOUBLE PRECISION* for mkl_domatadd_batch_strided, **COMPLEX** for mkl_comatadd_batch_strided, *DOUBLE COMPLEX* for mkl_zomatadd_batch_strided. Scaling factor for the matrices A.
a	**REAL** for mkl_somatadd_batch_strided, *DOUBLE PRECISION* for mkl_domatadd_batch_strided, **COMPLEX** for mkl_comatadd_batch_strided, *DOUBLE COMPLEX* for mkl_zomatadd_batch_strided. Array holding the input matrices A. Must have size at least stride_a*batch_size.
lda	**INTEGER** Leading dimension of the A matrices. If matrices are stored using column major layout, lda must be at least rows if A is not transposed or cols if A is transposed. If matrices are stored using row major layout, lda must be at least cols if A is not transposed or at least rows if A is transposed. Must be positive.
stride_a	**INTEGER** Stride between the different A matrices. If matrices are stored using column major layout, stride_a must be at least lda*rows if A is not transposed or at least lda*cols if A is transposed. If matrices are stored using row major layout, stride_a must be at least lda*rows if B is not transposed or at least lda*cols if A is transposed.
beta	**REAL** for mkl_somatadd_batch_strided, *DOUBLE PRECISION* for mkl_domatadd_batch_strided, **COMPLEX** for mkl_comatadd_batch_strided, *DOUBLE COMPLEX* for mkl_zomatadd_batch_strided. Scaling factor for the matrices B.
b	**REAL** for mkl_somatadd_batch_strided, *DOUBLE PRECISION* for mkl_domatadd_batch_strided, **COMPLEX** for mkl_comatadd_batch_strided, *DOUBLE COMPLEX* for mkl_zomatadd_batch_strided. Array holding the input matrices B. Must have size at least stride_b*batch_size.

ldb	**INTEGER** Leading dimension of the B matrices. If matrices are stored using column major layout, `ldb` must be at least `rows` if B is not transposed or `cols` if B is transposed. If matrices are stored using row major layout, `ldb` must be at least `cols` if B is not transposed or at least `rows` if B is transposed. Must be positive.
stride_b	**INTEGER** Stride between the different B matrices. If matrices are stored using column major layout, `stride_b` must be at least `ldb*cols` if B is not transposed or at least `ldb*rows` if B is transposed. If matrices are stored using row major layout, `stride_b` must be at least `ldb*rows` if B is not transposed or at least `ldb*cols` if B is transposed.
c	**REAL** for `mkl_somatadd_batch_strided`, *DOUBLE PRECISION* for `mkl_domatadd_batch_strided`, **COMPLEX** for `mkl_comatadd_batch_strided`, *DOUBLE COMPLEX* for `mkl_zomatadd_batch_strided`. Output array, overwritten by `batch_size` matrix addition operations of the form `alpha*op(A) + beta*op(B)`. Must have size at least `stride_c*batch_size`.
ldc	**INTEGER** Leading dimension of the A matrices. If matrices are stored using column major layout, `lda` must be at least `rows`. If matrices are stored using row major layout, `lda` must be at least `cols`. Must be positive.
stride_c	**INTEGER** Stride between the different C matrices. If matrices are stored using column major layout, `stride_c` must be at least `ldc*cols`. If matrices are stored using row major layout, `stride_c` must be at least `ldc*rows`.
batch_size	**INTEGER** Specifies the number of input and output matrices to add.

Output Parameters

c	Array holding the updated matrices `C`.

mkl_?omatcopy

Performs scaling and out-place transposition/copying of matrices.

Syntax

```
call mkl_somatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
call mkl_domatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
call mkl_comatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
call mkl_zomatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
```

Include Files

- `mkl.fi`

Description

The `mkl_?omatcopy` routine performs scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

B := *alpha**op(*A*)

The routine parameter descriptions are common for all implemented interfaces with the exception of data types that mostly refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

NOTE
Different arrays must not overlap.

Input Parameters

ordering — CHARACTER*1. Ordering of the matrix storage.

If *ordering* = 'R' or 'r', the ordering is row-major.

If *ordering* = 'C' or 'c', the ordering is column-major.

trans — CHARACTER*1. Parameter that specifies the operation type.

If *trans* = 'N' or 'n', op(*A*)=*A* and the matrix *A* is assumed unchanged on input.

If *trans* = 'T' or 't', it is assumed that *A* should be transposed.

If *trans* = 'C' or 'c', it is assumed that *A* should be conjugate transposed.

If *trans* = 'R' or 'r', it is assumed that *A* should be only conjugated.

If the data is real, then *trans* = 'R' is the same as *trans* = 'N', and *trans* = 'C' is the same as *trans* = 'T'.

rows — INTEGER. The number of rows in matrix *A* (the input matrix).

cols — INTEGER. The number of columns in matrix *A* (the input matrix).

alpha — REAL for mkl_somatcopy.

DOUBLE PRECISION for mkl_domatcopy.

COMPLEX for mkl_comatcopy.

DOUBLE COMPLEX for mkl_zomatcopy.

This parameter scales the input matrix by *alpha*.

a — REAL for mkl_somatcopy.

DOUBLE PRECISION for mkl_domatcopy.

COMPLEX for mkl_comatcopy.

DOUBLE COMPLEX for mkl_zomatcopy.

Input array.

If *ordering* = 'R' or 'r', the size of *a* is *lda***rows*.

If *ordering* = 'C' or 'c', the size of *a* is *lda***cols*.

lda — INTEGER. (Fortran interface).

If *ordering* = 'R' or 'r', *lda* represents the number of elements in array *a* between adjacent rows of matrix *A*; *lda* must be at least equal to the number of columns of matrix *A*.

If *ordering* = 'C' or 'c', *lda* represents the number of elements in array *a* between adjacent columns of matrix *A*; *lda* must be at least equal to the number of row in matrix *A*.

b

REAL for mkl_somatcopy.

DOUBLE PRECISION for mkl_domatcopy.

COMPLEX for mkl_comatcopy.

DOUBLE COMPLEX for mkl_zomatcopy.

Output array.

If *ordering* = 'R' or 'r';

- If *trans* = 'T' or 't' or 'C' or 'c', the size of *b* is *ldb* * *cols*.
- If *trans* = 'N' or 'n' or 'R' or 'r', the size of *b* is *ldb* * *rows*.

If *ordering* = 'C' or 'c';

- If *trans* = 'T' or 't' or 'C' or 'c', the size of *b* is *ldb* * *rows*.
- If *trans* = 'N' or 'n' or 'R' or 'r', the size of *b* is *ldb* * *cols*.

ldb

INTEGER. (Fortran interface).

If *ordering* = 'R' or 'r', *ldb* represents the number of elements in array *b* between adjacent rows of matrix *B*.

- If *trans* = 'T' or 't' or 'C' or 'c', *ldb* must be at least equal to *rows*.
- If *trans* = 'N' or 'n' or 'R' or 'r', *ldb* must be at least equal to *cols*.

If *ordering* = 'C' or 'c', *ldb* represents the number of elements in array *b* between adjacent columns of matrix *B*.

- If *trans* = 'T' or 't' or 'C' or 'c', *ldb* must be at least equal to *cols*.
- If *trans* = 'N' or 'n' or 'R' or 'r', *ldb* must be at least equal to *rows*.

Output Parameters

b

REAL for mkl_somatcopy.

DOUBLE PRECISION for mkl_domatcopy.

COMPLEX for mkl_comatcopy.

DOUBLE COMPLEX for mkl_zomatcopy.

Output array.

Contains the destination matrix.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_somatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, lda, ldb
   REAL alpha, b(ldb,*), a(lda,*)
```

```
SUBROUTINE mkl_domatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, lda, ldb
   DOUBLE PRECISION alpha, b(ldb,*), a(lda,*)
```

```
SUBROUTINE mkl_comatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, lda, ldb
   COMPLEX alpha, b(ldb,*), a(lda,*)
```

```
SUBROUTINE mkl_zomatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, lda, ldb
   DOUBLE COMPLEX alpha, b(ldb,*), a(lda,*)
```

mkl_?omatcopy_batch

Computes a group of out of place scaled matrix copy or transposition operations on general matrices.

Syntax

`call mkl_somatcopy_batch(`*layout*`, `*trans_array*`, `*rows_array*`, `*cols_array*`, `*alpha_array*`, `*A_array*`, `*lda_array*`, `*B_array*`, `*ldb_array*`, `*group_count*`, `*group_size*`)`

`call mkl_domatcopy_batch(`*layout*`, `*trans_array*`, `*rows_array*`, `*cols_array*`, `*alpha_array*`, `*A_array*`, `*lda_array*`, `*B_array*`, `*ldb_array*`, `*group_count*`, `*group_size*`)`

`call mkl_comatcopy_batch(`*layout*`, `*trans_array*`, `*rows_array*`, `*cols_array*`, `*alpha_array*`, `*A_array*`, `*lda_array*`, `*B_array*`, `*ldb_array*`, `*group_count*`, `*group_size*`)`

`call mkl_zomatcopy_batch(`*layout*`, `*trans_array*`, `*rows_array*`, `*cols_array*`, `*alpha_array*`, `*A_array*`, `*lda_array*`, `*B_array*`, `*ldb_array*`, `*group_count*`, `*group_size*`)`

Description

The `mkl_?omatcopy_batch` routine performs a series of out-of-place scaled matrix copies or transpositions. They are similar to the `mkl_?omatcopy` routine counterparts, but the `mkl_?omatcopy_batch` routine performs matrix operations with groups of matrices. Each group has the same parameters (matrix size, leading dimension, and scaling parameter), but a single call to `mkl_?omatcopy_batch` operates on multiple groups, and each group can have different parameters, unlike the related `mkl_?omatcopy_batch_strided` routines.

The operation is defined as

```
idx = 0
for i = 0..group_count - 1
     m in rows_array[i], n in cols_array[i], and alpha in alpha_array[i]
     for j = 0..group_size[i] - 1
          A and B matrices in a_array[idx] and b_array[idx], respectively
          B := alpha*op(A)
```

```
          idx = idx + 1
      end for
end for
```

Where op(*X*) is one of op(*X*)=*X*, op(*X*)=*X*', op(*X*)=conjg(*X*'), or op(*X*)=conjg(*X*). *A* is a *m*-by-*n* matrix such that *m* and *n* are elements of *rows_array* and *cols_array*.

A and *B* represent matrices stored at addresses pointed to by *A_array* and *B_array*. The number of entries in *A_array* and *B_array* is *total_batch_count* = the sum of all of the *group_size* entries.

Input Parameters

layout

CHARACTER*1.

Specifies whether two-dimensional array storage is row-major (R) or column-major (C).

trans_array

CHARACTER*1.

Array of size *group_count*. For the group *i*, *trans* = *trans_array*[*i*] specifies the form of op(*A*), the transposition operation applied to the *A* matrix:

If *trans* = 'N' or 'n', op(*A*)=*A*.

If *trans* = 'T' or 't', op(*A*)=*A*'

If *trans* = 'C' or 'c', op(*A*)=*conjg(A')*

If *trans* = 'R' or 'r', op(*A*)=*conjg(A)*

rows_array

INTEGER. Array of size *group_count*. Specifies the number of rows of the matrix *A*. The value of each element must be at least zero.

cols_array

INTEGER. Array of size *group_count*. Specifies the number of columns of the matrix *A*. The value of each element must be at least zero.

alpha_array

REAL for mkl_somatcopy_batch.

DOUBLE PRECISION for mkl_domatcopy_batch.

COMPLEX for mkl_comatcopy_batch.

DOUBLE COMPLEX for mkl_zomatcopy_batch.

Array of size *group_count*. Specifies the scalar *alpha*.

A_array

INTEGER*8 for Intel® 64 architecture.

INTEGER*4 for IA-32 architecture.

Array of size *total_batch_count*, holding pointers to arrays used to store *A* input matrices.

lda_array

INTEGER. Array of size *group_count*. The leading dimension of the input matrix A. It must be positive and at least *m* if column major layout is used or at least *n* if row major layout is used.

ldb_array

INTEGER. Array of size *group_count*. The leading dimension of the output matrix B. It must be positive and at least

m if column major layout is used and op(A) = A or conjg(A)

n if row major layout is used and op(A) = A' or conjg(A')

	n otherwise
group_count	INTEGER. Specifies the number of groups. Must be at least 0
group_size	INTEGER. Array of size *group_count*. The element *group_size*[*i*] specifies the number of matrices in group *i*. Each element in *group_size* must be at least 0.

Output Parameters

B_array	INTEGER*8 for Intel® 64 architecture. INTEGER*4 for IA-32 architecture. Output array of size *total_batch_count*, holding pointers to arrays used to store the *B* output matrices, the contents of which are overwritten by the operation of the form alpha*op(*A*).

mkl_?omatcopy_batch_strided

Computes a group of out of place scaled matrix copy or transposition using general matrices.

Syntax

```
call mkl_somatcopy_batch_strided(layout, trans, row, col, alpha, a, lda, stridea, b, ldb, strideb, batch_size)
call mkl_domatcopy_batch_strided(layout, trans, row, col, alpha, a, lda, stridea, b, ldb, strideb, batch_size)
call mkl_comatcopy_batch_strided(layout, trans, row, col, alpha, a, lda, stridea, b, ldb, strideb, batch_size)
call mkl_zomatcopy_batch_strided(layout, trans, row, col, alpha, a, lda, stridea, b, ldb, strideb, batch_size)
```

Description

The mkl_?omatcopy_batch_strided routine performs a series of out-of-place scaled matrix copy or transposition. They are similar to the mkl_?omatcopy routine counterparts, but the mkl_?omatcopy_batch_strided routine performs matrix operations with group of matrices.

All matrices a and b have the same parameters (size, transposition operation…) and are stored at constant stride from each other respectively given by *stridea* and *strideb*. The operation is defined as

```
for i = 0 … batch_size – 1
    A and B are matrices at offset i * stridea in a and I * strideb in b
    B = alpha * op(A)
end for
```

Input Parameters

layout	CHARACTER*1. Specifies whether two-dimensional array storage is row-major or column-major .
trans	CHARACTER*1. Specifies op(A), the transposition operation applied to the AB matrices. If *trans* = 'N' or 'n', op(*A*)=*A*.

If *trans* = 'T' or 't', op(*A*)=*A'*

If *trans* = 'C' or 'c', op(*A*)=*conig(A')*

If *trans* = 'R' or 'r', op(*A*)=*conig(A)*

row — INTEGER. Specifies the number of rows of the matrices A and B. The value of *row* must be at least zero.

col — INTEGER. Specifies the number of columns of the matrices A and B. The value of *col* must be at least zero.

alpha — REAL for mkl_somatcopy_batch_strided.

DOUBLE PRECISION for mkl_domatcopy_batch_strided.

COMPLEX for mkl_comatcopy_batch_strided.

DOUBLE COMPLEX for mkl_zomatcopy_batch_strided.

Specifies the scalar *alpha*.

a — REAL for mkl_somatcopy_batch_strided.

DOUBLE PRECISION for mkl_domatcopy_batch_strided.

COMPLEX for mkl_comatcopy_batch_strided.

DOUBLE COMPLEX for mkl_zomatcopy_batch_strided.

Array holding all the input matrices A. Must be of size at least lda * k + stridea * (batch_size - 1) * stridea where k is *col* if column major is used and *row* otherwise.

lda — INTEGER.

The leading dimension of the matrix input A. It must be positive and at least *row* if column major layout is used or at least *col* if row major layout is used.

stridea — INTEGER. Stride between two consecutive A matrices, must be at least 0.

b — REAL for mkl_somatcopy_batch_strided.

DOUBLE PRECISION for mkl_domatcopy_batch_strided.

COMPLEX for mkl_comatcopy_batch_strided.

DOUBLE COMPLEX for mkl_zomatcopy_batch_strided.

Array holding all the output matrices B. Must be of size at least batch_size * strideb. The b array must be independent from the a array.

ldb — INTEGER.

The leading dimension of the output matrix B. It must be positive and at least:

- *row* if column major layout is used and op(A) = A or conjg(A)
- *row* if row major layout is used and op(A) = A' or conjg(A')
- *col* otherwise

strideb — INTEGER.

Stride between two consecutive B matrices. It must be positive and at least:

- *ldb** *col* if column major layout is used and op(A) = A or conjg(A)
- *ldb** *col* if row major layout is used and op(A) = A' or conjg(A')
- *ldb***row* otherwise

batch_size

Output Parameters

b Array holding the batch_size updated matrices B.

mkl_?omatcopy2

Performs two-strided scaling and out-of-place transposition/copying of matrices.

Syntax

call mkl_somatcopy2(*ordering*, *trans*, *rows*, *cols*, *alpha*, *a*, *lda*, *stridea*, *b*, *ldb*, *strideb*)

call mkl_domatcopy2(*ordering*, *trans*, *rows*, *cols*, *alpha*, *a*, *lda*, *stridea*, *b*, *ldb*, *strideb*)

call mkl_comatcopy2(*ordering*, *trans*, *rows*, *cols*, *alpha*, *a*, *lda*, *stridea*, *b*, *ldb*, *strideb*)

call mkl_zomatcopy2(*ordering*, *trans*, *rows*, *cols*, *alpha*, *a*, *lda*, *stridea*, *b*, *ldb*, *strideb*)

Include Files

- mkl.fi

Description

The mkl_?omatcopy2 routine performs two-strided scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

B := *alpha**op(*A*)

Normally, matrices in the BLAS or LAPACK are specified by a single stride index. For instance, in the column-major order, *A*(2,1) is stored in memory one element away from *A*(1,1), but *A*(1,2) is a leading dimension away. The leading dimension in this case is at least the number of rows of the source matrix. If a matrix has two strides, then both *A*(2,1) and *A*(1,2) may be an arbitrary distance from *A*(1,1).

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

NOTE
Different arrays must not overlap.

Input Parameters

ordering

CHARACTER*1. Ordering of the matrix storage.

If *ordering* = 'R' or 'r', the ordering is row-major.

If *ordering* = 'C' or 'c', the ordering is column-major.

trans

CHARACTER*1. Parameter that specifies the operation type.

If *trans* = 'N' or 'n', op(*A*)=*A* and the matrix *A* is assumed unchanged on input.

If *trans* = 'T' or 't', it is assumed that *A* should be transposed.

If *trans* = 'C' or 'c', it is assumed that *A* should be conjugate transposed.

If *trans* = 'R' or 'r', it is assumed that *A* should be only conjugated.

If the data is real, then *trans* = 'R' is the same as *trans* = 'N', and *trans* = 'C' is the same as *trans* = 'T'.

rows

INTEGER. number of rows for the input matrix *A*. Must be at least zero.

cols

INTEGER. Number of columns for the input matrix *A*. Must be at least zero.

alpha

REAL for mkl_somatcopy2.

DOUBLE PRECISION for mkl_domatcopy2.

COMPLEX for mkl_comatcopy2.

DOUBLE COMPLEX for mkl_zomatcopy2.

Scaling factor for the matrix transposition or copy.

a

REAL for mkl_somatcopy2.

DOUBLE PRECISION for mkl_domatcopy2.

COMPLEX for mkl_comatcopy2.

DOUBLE COMPLEX for mkl_zomatcopy2.

Array holding the input matrix *A*. Must have size at least *lda* * *n* for column major ordering and at least *lda* * *m* for row major ordering.

lda

INTEGER.

Leading dimension of the matrix *A*. If matrices are stored using column major layout, *lda* is the number of elements in the array between adjacent columns of the matrix and must be at least *stridea* * (*m*-1) + 1. If using row major layout, *lda* is the number of elements between adjacent rows of the matrix and must be at least *stridea* * (*n*-1) + 1.

stridea

INTEGER.

The second stride of the matrix *A*. For column major layout, *stridea* is the number of elements in the array between adjacent rows of the matrix. For row major layout *stridea* is the number of elements between adjacent columns of the matrix. In both cases *stridea* must be at least 1.

b

REAL for mkl_somatcopy2.

`DOUBLE PRECISION` for `mkl_domatcopy2`.

`COMPLEX` for `mkl_comatcopy2`.

`DOUBLE COMPLEX` for `mkl_zomatcopy2`.

Array holding the output matrix *B*.

	trans* = *transpose::nontrans	***trans* = *transpose::trans*, or *trans* = *transpose::conjtrans***
Column major	*B* is *m* x *n* matrix. Size of array *b* must be at least *ldb* * *n*.	*B* is *n* x *m* matrix. Size of array *b* must be at least *ldb* * *m*.
Row major	*B* is *m* x *n* matrix. Size of array *b* must be at least *ldb* * *m*.	*B* is *n* x *m* matrix. Size of array *b* must be at least *ldb* * *n*.

ldb

`INTEGER`.

The leading dimension of the matrix *B*. Must be positive.

	trans* = *transpose::nontrans	***trans* = *transpose::trans*, or *trans* = *transpose::conjtrans***
Column major	*ldb* must be at least *strideb* * (*m*-1) + 1.	*ldb* must be at least *strideb* * (*n*-1) + 1.
Row major	*ldb* must be at least *strideb* * (*n*-1) + 1.	*ldb* must be at least *strideb* * (*m*-1) + 1.

strideb

`INTEGER`.

The second stride of the matrix *B*. For column major layout, *strideb* is the number of elements in the array between adjacent rows of the matrix. For row major layout, *strideb* is the number of elements between adjacent columns of the matrix. In both cases *strideb* must be at least 1.

Output Parameters

b

`REAL` for `mkl_somatcopy2`.

`DOUBLE PRECISION` for `mkl_domatcopy2`.

`COMPLEX` for `mkl_comatcopy2`.

`DOUBLE COMPLEX` for `mkl_zomatcopy2`.

Array, size at least *m*.

Contains the destination matrix.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_somatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb,
strideb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, lda, stridea, ldb, strideb
   REAL  alpha, b(*), a(*)
```

```
SUBROUTINE mkl_domatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb,
strideb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, lda, stridea, ldb, strideb
   DOUBLE PRECISION alpha, b(*), a(*)
```

```
SUBROUTINE mkl_comatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb,
strideb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, lda, stridea, ldb, strideb
   COMPLEX alpha, b(*), a(*)
```

```
SUBROUTINE mkl_zomatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb,
strideb )
   CHARACTER*1 ordering, trans
   INTEGER rows, cols, lda, stridea, ldb, strideb
   DOUBLE COMPLEX alpha, b(*), a(*)
```

mkl_?omatadd

Scales and sums two matrices including in addition to performing out-of-place transposition operations.

Syntax

`call mkl_somatadd(`*`ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc`*`)`

`call mkl_domatadd(`*`ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc`*`)`

`call mkl_comatadd(`*`ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc`*`)`

`call mkl_zomatadd(`*`ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc`*`)`

Include Files

- `mkl.fi`

Description

The `mkl_?omatadd` routine scales and adds two matrices, as well as performing out-of-place transposition operations. A transposition operation can be no operation, a transposition, a conjugate transposition, or a conjugation (without transposition). The following out-of-place memory movement is done:

```
C := alpha*op(A) + beta*op(B)
```

where the `op(A)` and `op(B)` operations are transpose, conjugate-transpose, conjugate (no transpose), or no transpose, depending on the values of *`transa`* and *`transb`*. If no transposition of the source matrices is required, *`m`* is the number of rows and *`n`* is the number of columns in the source matrices *A* and *B*. In this case, the output matrix *C* is *m*-by-*n*.

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

NOTE
Note that different arrays must not overlap.

Input Parameters

ordering
CHARACTER*1. Ordering of the matrix storage.
If *ordering* = 'R' or 'r', the ordering is row-major.
If *ordering* = 'C' or 'c', the ordering is column-major.

transa
CHARACTER*1. Parameter that specifies the operation type on matrix *A*.
If *transa* = 'N' or 'n', op(*A*)=*A* and the matrix *A* is assumed unchanged on input.
If *transa* = 'T' or 't', it is assumed that *A* should be transposed.
If *transa* = 'C' or 'c', it is assumed that *A* should be conjugate transposed.
If *transa* = 'R' or 'r', it is assumed that *A* should be conjugated (and not transposed).
If the data is real, then *transa* = 'R' is the same as *transa* = 'N', and *transa* = 'C' is the same as *transa* = 'T'.

transb
CHARACTER*1. Parameter that specifies the operation type on matrix *B*.
If *transb* = 'N' or 'n', op(*B*)=*B* and the matrix *B* is assumed unchanged on input.
If *transb* = 'T' or 't', it is assumed that *B* should be transposed.
If *transb* = 'C' or 'c', it is assumed that *B* should be conjugate transposed.
If *transb* = 'R' or 'r', it is assumed that *B* should be conjugated (and not transposed).
If the data is real, then *transb* = 'R' is the same as *transb* = 'N', and *transb* = 'C' is the same as *transb* = 'T'.

m
INTEGER. The number of matrix rows in op(*A*), op(*B*), and *C*.

n
INTEGER. The number of matrix columns in op(*A*), op(*B*), and *C*.

alpha
REAL for mkl_somatadd.
DOUBLE PRECISION for mkl_domatadd.
COMPLEX for mkl_comatadd.
DOUBLE COMPLEX for mkl_zomatadd.
This parameter scales the input matrix by *alpha*.

a
REAL for mkl_somatadd.

DOUBLE PRECISION for mkl_domatadd.

COMPLEX for mkl_comatadd.

DOUBLE COMPLEX for mkl_zomatadd.

Array, size *a*(*lda*,*).

lda

INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix *A*; measured in the number of elements.

For *ordering* = 'C' or 'c': when *transa* = 'N', 'n', 'R', or 'r', *lda* must be at least max(1,*m*); otherwise *lda* must be max(1,*n*).

For *ordering* = 'R' or 'r': when *transa* = 'N', 'n', 'R', or 'r', *lda* must be at least max(1,*n*); otherwise *lda* must be max(1,*m*).

beta

REAL for mkl_somatadd.

DOUBLE PRECISION for mkl_domatadd.

COMPLEX for mkl_comatadd.

DOUBLE COMPLEX for mkl_zomatadd.

This parameter scales the input matrix by *beta*.

b

REAL for mkl_somatadd.

DOUBLE PRECISION for mkl_domatadd.

COMPLEX for mkl_comatadd.

DOUBLE COMPLEX for mkl_zomatadd.

Array, size *b*(*ldb*,*).

ldb

INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix *B*; measured in the number of elements.

For *ordering* = 'C' or 'c': when *transa* = 'N', 'n', 'R', or 'r', *ldb* must be at least max(1,*m*); otherwise *ldb* must be max(1,*n*).

For *ordering* = 'R' or 'r': when *transa* = 'N', 'n', 'R', or 'r', *ldb* must be at least max(1,*n*); otherwise *ldb* must be max(1,*m*).

ldc

INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix *C*; measured in the number of elements.

If *ordering* = 'C' or 'c', then *ldc* must be at least max(1, *m*), otherwise *ldc* must be at least max(1, *n*).

Output Parameters

c

REAL for mkl_somatadd.

DOUBLE PRECISION for mkl_domatadd.

COMPLEX for mkl_comatadd.

DOUBLE COMPLEX for mkl_zomatadd.

Array, size *c*(*ldc*,*).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_somatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )
   CHARACTER*1 ordering, transa, transb
   INTEGER m, n, lda, ldb, ldc
   REAL alpha, beta
   REAL a(lda,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_domatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )
   CHARACTER*1 ordering, transa, transb
   INTEGER m, n, lda, ldb, ldc
   DOUBLE PRECISION alpha, beta
   DOUBLE PRECISION a(lda,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_comatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )
   CHARACTER*1 ordering, transa, transb
   INTEGER m, n, lda, ldb, ldc
   COMPLEX alpha, beta
   COMPLEX a(lda,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zomatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )
   CHARACTER*1 ordering, transa, transb
   INTEGER m, n, lda, ldb, ldc
   DOUBLE COMPLEX alpha, beta
   DOUBLE COMPLEX a(lda,*), b(ldb,*), c(ldc,*)
```

?gemm_pack_get_size, gemm_*_pack_get_size

Returns the number of bytes required to store the packed matrix.

Syntax

`dest = sgemm_pack_get_size (identifier, m, n, k)`

`dest = dgemm_pack_get_size (identifier, m, n, k)`

`dest = gemm_s8u8s32_pack_get_size (identifier, m, n, k)`

`dest = gemm_s16s16s32_pack_get_size (identifier, m, n, k)`

Include Files

- `mkl.fi`

Description

The `?gemm_pack_get_size` and `gemm_*_pack_get_size` routines belong to a set of related routines that enable the use of an internal packed storage. Call the `?gemm_pack_get_size` and `gemm_*_pack_get_size` routines first to query the size of storage required for a packed matrix structure to be used in subsequent calls. Ultimately, the packed matrix structure is used to compute

C := *alpha**op(*A*)*op(*B*) + *beta***C* for bfloat16, half, single and double precision or

C := *alpha**(op(*A*)+ *A_offset*)*(op(*B*)+ *B_offset*) + *beta***C* + *C_offset* for integer type.

where:

op(*X*) is one of the operations op(*X*) = *X* or op(*X*) = X^T
alpha and *beta* are scalars,
A , *A_offset*,*B*, *B_offset*,*C*, and *C_offset* are matrices
op(*A*) is an *m*-by-*k* matrix,
op(*B*) is a *k*-by-*n* matrix,
C is an *m*-by-*n* matrix.
A_offset is an *m*-by-*k* matrix.
B_offset is an *k*-by-*n* matrix.
C_offset is an *m*-by-*n* matrix.

Input Parameters

Parameter	Type	Description
identifier	CHARACTER*1.	Specifies which matrix is to be packed: If *identifier* = 'A' or 'a', the size returned is the size required to store matrix *A* in an internal format. If *identifier* = 'B' or 'b', the size returned is the size required to store matrix *B* in an internal format.
m	INTEGER.	Specifies the number of rows of matrix op(*A*) and of the matrix *C*. The value of *m* must be at least zero.
n	INTEGER.	Specifies the number of columns of matrix op(*B*) and the number of columns of matrix *C*. The value of *n* must be at least zero.
k	INTEGER.	Specifies the number of columns of matrix op(*A*) and the number of rows of matrix op(*B*). The value of *k* must be at least zero.

Return Values

Parameter	Type	Description
size	INTEGER.	Returns the size (in bytes) required to store the matrix when packed into the internal format of Intel® oneAPI Math Kernel Library (oneMKL).

Example

See the following examples in the MKL installation directory to understand the use of these routines:

sgemm_pack_get_size: examples\blas\source\sgemm_computex.f

dgemm_pack_get_size: examples\blas\source\dgemm_computex.f

gemm_s8u8s32_pack_get_size: examples\blas\source\gemm_s8u8s32_computex.f

gemm_s16s16s32_pack_get_size: examples\blas\source\gemm_s16s16s32_computex.f

See Also

?gemm_pack and gemm_*_pack

to pack the matrix into a buffer allocated previously.

`?gemm_compute` and `?gemm_*_compute`
to compute a matrix-matrix product with general matrices (where one or both input matrices are stored in a packed data structure) and add the result to a scalar-matrix product.

?gemm_pack

Performs scaling and packing of the matrix into the previously allocated buffer.

Syntax

```
call sgemm_pack (identifier, trans, m, n, k, alpha, src, ld, dest)
call dgemm_pack (identifier, trans, m, n, k, alpha, src, ld, dest)
```

Include Files

- mkl.fi

Description

The `?gemm_pack` routine is one of a set of related routines that enable use of an internal packed storage. Call `?gemm_pack` after you allocate a buffer whose size is given by `?gemm_pack_get_size`. The `?gemm_pack` routine scales the identified matrix by alpha and packs it into the buffer allocated previously.

> **NOTE**
> Do not copy the packed matrix to a different address because the internal implementation depends on the alignment of internally-stored metadata.

The `?gemm_pack` routine performs this operation:

dest := *alpha**op(*src*) as part of the computation *C* := *alpha**op(*A*)*op(*B*) + *beta***C*

where:

op(*X*) is one of the operations op(*X*) = *X*, op(*X*) = X^T, or op(*X*) = X^H,
alpha and *beta* are scalars,
src is a matrix,
A , *B*, and *C* are matrices
op(*src*) is an *m*-by-*k* matrix if *identifier* = 'A' or 'a',
op(*src*) is a *k*-by-*n* matrix if *identifier* = 'B' or 'b',
dest is an internal packed storage buffer.

> **NOTE**
> For best performance, use the same number of threads for packing and for computing.
>
> If packing for both *A* and *B* matrices, you must use the same number of threads for packing *A* as for packing *B*.

Input Parameters

identifier — CHARACTER*1. Specifies which matrix is to be packed:

If *identifier* = 'A' or 'a', the routine allocates storage to pack matrix *A*.

If *identifier* = 'B' or 'b', the routine allocates storage to pack matrix *B*.

trans

CHARACTER*1. Specifies the form of op(*src*) used in the packing:

If *trans* = 'N' or 'n' op(*src*) = *src*.

If *trans* = 'T' or 't' op(*src*) = src^T.

If *trans* = 'C' or 'c' op(*src*) = src^H.

m

INTEGER. Specifies the number of rows of the matrix op(*A*) and of the matrix *C*. The value of *m* must be at least zero.

n

INTEGER. Specifies the number of columns of the matrix op(*B*) and the number of columns of the matrix *C*. The value of *n* must be at least zero.

k

INTEGER. Specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*). The value of *k* must be at least zero.

alpha

REAL for sgemm_pack

DOUBLE PRECISION for dgemm_pack

Specifies the scalar *alpha*.

src

REAL for sgemm_pack

DOUBLE PRECISION for dgemm_pack

Array:

	trans = 'N' or 'n'	*trans* = 'T', 't', 'C', or 'c'
identifier = 'A' or 'a'	Size *ld***k*. Before entry, the leading *m*-by-*k* part of the array *src* must contain the matrix *A*.	Size *ld***m*. Before entry, the leading *k*-by-*m* part of the array *src* must contain the matrix *A*.
identifier = 'B' or 'b'	Size *ld***n*. Before entry, the leading *k*-by-*n* part of the array *src* must contain the matrix *B*.	Size *ld***k*. Before entry, the leading *n*-by-*k* part of the array *src* must contain the matrix *B*.

ld

INTEGER. Specifies the leading dimension of *src* as declared in the calling (sub)program.

	trans = 'N' or 'n'	*trans* = 'T', 't', 'C', or 'c'
identifier = 'A' or 'a'	*ld* must be at least max(1, *m*).	*ld* must be at least max(1, *k*).

identifier = 'B' or 'b'	*ld* must be at least max(1, *k*).	*ld* must be at least max(1, *n*).

dest POINTER.

Scaled and packed internal storage buffer.

Output Parameters

dest Overwritten by the matrix *alpha**op(*src*).

See Also

?gemm_pack_get_size Returns the number of bytes required to store the packed matrix.

?gemm_compute Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.

?gemm
for a detailed description of general matrix multiplication.

gemm_*_pack

Pack the matrix into the buffer allocated previously.

Syntax

```
call gemm_s8u8s32_pack (identifier, trans, m, n, k, src, ld, dest)
call gemm_s16s16s32_pack (identifier, trans, m, n, k, src, ld, dest)
```

Include Files

- mkl.fi

Description

The gemm_*_pack routine is one of a set of related routines that enable the use of an internal packed storage. Call gemm_*_pack after you allocate a buffer whose size is given by gemm_*_pack_get_size. The gemm_*_pack routine packs the identified matrix into the buffer allocated previously.

The gemm_*_pack routine performs this operation:

dest := op(*src*) as part of the computation *C* := *alpha**(op(*A*) + *A_offset*)*(op(*B*) + *B_offset*) + *beta***C* + *C_offset* for integer types.

C := *alpha**op(*A*) * op(*B*) + *beta***C* for bfloat16 type.

where:

op(*X*) is one of the operations op(*X*) = *X* or op(*X*) = X^T
alpha and *beta* are scalars,
src is a matrix,
A , *A_offset*,*B*, *B_offset*,*c*,and *C_offset* are matrices
op(*src*) is an *m*-by-*k* matrix if *identifier* = 'A' or 'a',
op(*src*) is a *k*-by-*n* matrix if *identifier* = 'B' or 'b',
dest is the buffer previously allocated to store the matrix packed into an internal format
A_offset is an *m*-by-*k* matrix.
B_offset is an *k*-by-*n* matrix.

C_offset is an *m*-by-*n* matrix.

NOTE
For best performance, use the same number of threads for packing and for computing.

If packing for both *A* and *B* matrices, you must use the same number of threads for packing *A* as for packing *B*.

Input Parameters

identifier

CHARACTER*1.

Specifies which matrix is to be packed:

If *identifier* = 'A' or 'a', the *A* matrix is packed.

If *identifier* = 'B' or 'b', the *B* matrix is packed.

trans

CHARACTER*1.

Specifies the form of op(*src*) used in the packing:

If *trans* = 'N' or 'n' op(*src*) = *src*.

If *trans* = 'T' or 't' op(*src*) = src^T.

m

INTEGER.

Specifies the number of rows of matrix op(*A*) and of the matrix *C*. The value of *m* must be at least zero.

n

INTEGER.

Specifies the number of columns of matrix op(*B*) and the number of columns of matrix *C*. The value of *n* must be at least zero.

k

INTEGER.

Specifies the number of columns of matrix op(*A*) and the number of rows of matrix op(*B*). The value of *k* must be at least zero.

src

INTEGER*1 for gemm_s8u8s32_pack and INTEGER*2 for gemm_s16s16s32_pack

	trans = 'N' or 'n'	*trans* = 'T' or 't'
identifier = 'A' or 'a'	Size *ld*k*. Before entry, the leading *m*-by-*k* part of the array *src* must contain the matrix *A*.	Size *ld*m*. Before entry, the leading *k*-by-*m* part of the array *src* must contain the matrix *A*.
identifier = 'B' or 'b'	Size *ld*n*. Before entry, the leading *k*-by-*n* part of the array *src* must contain the matrix *B*.	Size *ld*k*. Before entry, the leading *n*-by-*k* part of the array *src* must contain the matrix *B*.

ld

INTEGER. Specifies the leading dimension of *src* as declared in the calling (sub)program.

	trans = 'N' or 'n'	*trans* = 'T' or 't'
identifier = 'A' or 'a'	*ld* must be at least max(1, *m*).	*ld* must be at least max(1, *k*).
identifier = 'B' or 'b'	*ld* must be at least max(1, *k*).	*ld* must be at least max(1, *n*).

dest INTEGER*1 for gemm_s8u8s32_pack or INTEGER*2 for gemm_s16s16s32_pack
Buffer for the packed matrix.

Output Parameters

dest INTEGER*1 for gemm_s8u8s32_pack or INTEGER*2 for gemm_s16s16s32_pack
Overwritten by the matrix op(*src*) stored in a format internal to Intel® oneAPI Math Kernel Library (oneMKL).

Example

See the following examples in the MKL installation directory to understand the use of these routines:

gemm_s8u8s32_pack: examples\blas\source\gemm_s8u8s32_computex.f

gemm_s16s16s32_pack: examples\blas\source\gemm_s16s16s32_computex.f

Application Notes

See Also

gemm_*_pack_get_size
to return the number of bytes needed to store the packed matrix.

gemm_*_compute
to compute a matrix-matrix product with general integer matrices (where one or both input matrices are stored in a packed data structure) and add the result to a scalar-matrix product.

?gemm_compute

Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.

Syntax

```
call sgemm_compute (transa, transb, m, n, k, a, lda, b, ldb, beta, C, ldc)
call dgemm_compute (transa, transb, m, n, k, a, lda, b, ldb, beta, C, ldc)
```

Include Files

- mkl.fi

Description

The ?gemm_compute routine is one of a set of related routines that enable use of an internal packed storage. After calling ?gemm_pack call ?gemm_compute to compute

C := op(*A*)*op(*B*) + *beta**C*,

where:

op(*X*) is one of the operations op(*X*) = *X*, op(*X*) = X^T, or op(*X*) = X^H,
beta is a scalar,
A , *B*, and *C* are matrices:
op(*A*) is an *m*-by-*k* matrix,
op(*B*) is a *k*-by-*n* matrix,
C is an *m*-by-*n* matrix.

NOTE
For best performance, use the same number of threads for packing and for computing.

If packing for both *A* and *B* matrices, you must use the same number of threads for packing *A* as for packing *B*.

Input Parameters

transa

CHARACTER*1. Specifies the form of op(*A*) used in the matrix multiplication:

If *transa* = 'N' or 'n' op(*A*) = *A*.

If *transa* = 'T' or 't' op(*A*) = A^T.

If *transa* = 'C' or 'c' op(*A*) = A^H.

If *transa* = 'P' or 'p' the matrix in array *a* is packed and *lda* is ignored.

transb

CHARACTER*1. Specifies the form of op(*B*) used in the matrix multiplication:

If *transb* = 'N' or 'n' op(*B*) = *B*.

If *transb* = 'T' or 't' op(*B*) = B^T.

If *transb* = 'C' or 'c' op(*B*) = B^H.

If *transb* = 'P' or 'p' the matrix in array *b* is packed and *ldb* is ignored.

m

INTEGER. Specifies the number of rows of the matrix op(*A*) and of the matrix *C*. The value of *m* must be at least zero.

n

INTEGER. Specifies the number of columns of the matrix op(*B*) and the number of columns of the matrix *C*. The value of *n* must be at least zero.

k

INTEGER. Specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*). The value of *k* must be at least zero.

a

REAL for sgemm_compute

DOUBLE PRECISION for dgemm_compute

Array:

transa = 'N' or 'n'	*transa* = 'T', 't', 'C', or 'c'	*transa* = 'P' or 'p'

Size *lda*k*. Before entry, the leading *m*-by-*k* part of the array *a* must contain the matrix *A*.	Size *lda*m*. Before entry, the leading *k*-by-*m* part of the array *a* must contain the matrix *A*.	Stored in internal packed format.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

If *transa* = 'N' or 'n', *lda* must be at least max (1, *m*).

If *transa* = 'T', 't', 'C', or 'c', *lda* must be at least max (1, *k*).

If *transa* = 'P' or 'p', *lda* is ignored.

b REAL for sgemm_compute

DOUBLE PRECISION for dgemm_compute

Array:

transb = 'N' or 'n'	*transb* = 'T', 't', 'C', or 'c'	*transb* = 'P' or 'p'
Size *ldb*n*. Before entry, the leading *k*-by-*n* part of the array *b* must contain the matrix *B*.	Size *ldb*k*. Before entry, the leading *n*-by-*k* part of the array *b* must contain the matrix *B*.	Stored in internal packed format.

ldb INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

If *transb* = 'N' or 'n', *ldb* must be at least max (1, *k*).

If *transb* = 'T', 't', 'C', or 'c', *ldb* must be at least max (1, *n*).

If *transb* = 'P' or 'p', *ldb* is ignored.

beta REAL for sgemm_compute

DOUBLE PRECISION for dgemm_compute

Specifies the scalar *beta*. When *beta* is equal to zero, then *c* need not be set on input.

c REAL for sgemm_compute

DOUBLE PRECISION for dgemm_compute

Array, size *ldc* by *n*. Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is equal to zero, in which case *c* need not be set on entry.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

The value of *ldc* must be at least max (1, *m*).

Output Parameters

c	Overwritten by the *m*-by-*n* matrix op(*A*)*op(*B*) + *beta**C*.

See Also

?gemm_pack_get_size Returns the number of bytes required to store the packed matrix.

?gemm_pack Performs scaling and packing of the matrix into the previously allocated buffer.

?gemm
for a detailed description of general matrix multiplication.

gemm_*_compute

Computes a matrix-matrix product with general integer matrices (where one or both input matrices are stored in a packed data structure) and adds the result to a scalar-matrix product.

Syntax

```
call gemm_s8u8s32_compute (transa, transb, offsetc, m, n, k, alpha, a, lda, oa, b, ldb, ob, beta, c, ldc, oc)
```

```
call gemm_s16s16s32_compute (transa, transb, offsetc, m, n, k, alpha, a, lda, oa, b, ldb, ob, beta, c, ldc, oc)
```

Include Files

- mkl.fi

Description

The gemm_*_compute routine is one of a set of related routines that enable use of an internal packed storage. After calling gemm_*_pack call gemm_*_compute to compute

C := *alpha**(op(*A*) + *A_offset*)*(op(*B*) + *B_offset*) + *beta**C* + *C_offset*,

where:

op(*X*) is either op(*X*) = *X* or op(*X*) = X^T

alpha and *beta*are scalars

A , *B*, and *C* are matrices:

op(*A*) is an *m*-by-*k* matrix,

op(*B*) is a *k*-by-*n* matrix,

C is an *m*-by-*n* matrix.

A_offset is an *m*-by-*k* matrix with every element equal to the value oa.

B_offset is an *k*-by-*n* matrix with every element equal to the value ob.

C_offset is an *m*-by-*n* matrix defined by the oc array as described in the description of the *offsetc* parameter.

NOTE

For best performance, use the same number of threads for packing and for computing.

If you are packing for both *A* and *B* matrices, you must use the same number of threads for packing *A* as for packing *B*.

Input Parameters

transa CHARACTER*1. Specifies the form of op(*A*) used in the packing:

If *transa* = 'N' or 'n' op(*A*) = *A*.

If *transa* = 'T' or 't' op(*A*) = A^T.

If *transa* = 'P' or 'p' the matrix in array *a* is packed into a format internal to Intel® oneAPI Math Kernel Library (oneMKL) and *lda* is ignored.

transb CHARACTER*1. Specifies the form of op(*B*) used in the packing:

If *transb* = 'N' or 'n' op(*B*) = *B*.

If *transb* = 'T' or 't' op(*B*) = B^T.

If *transb* = 'P' or 'p' the matrix in array *b* is packed into a format internal to Intel® oneAPI Math Kernel Library (oneMKL) and *ldb* is ignored.

offsetc CHARACTER*1. Specifies the form of *C_offset* used in the matrix multiplication.

If *offsetc*='F' or 'f' :oc has a single element and every element of *C_offset* is equal to this element.

If *offsetc*='C' or 'c' :oc has a size of *m* and every element of *C_offset* is equal to oc.

If *offsetc*='R' or 'r' :oc has a size of *n* and every element of *C_offset* is equal to oc.

m INTEGER. Specifies the number of rows of the matrix op(*A*) and of the matrix *C*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of the matrix op(*B*) and the number of columns of the matrix *C*. The value of *n* must be at least zero.

k INTEGER. Specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*). The value of *k* must be at least zero.

alpha REAL. Specifies the scalar *alpha*.

a INTEGER*1 for gemm_s8u8s32_compute

INTEGER*2 for gemm_s16s16s32_compute

transa = 'N' or 'n'	*transa* = 'T', 't'	*transa* = 'P' or 'p'
Array, size *lda***k*. Before entry, the leading *m*-by-*k* part of the array *a* must contain the matrix *A*.	Array, size *lda***m*. Before entry, the leading *k*-by-*m* part of the array *a* must contain the matrix *A*.	Array of size returned by gemm_*_pack_get_size and initialized using gemm_*_pack

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

If *transa* = 'N' or 'n', *lda* must be at least max (1, *m*).

If *transa* = 'T', 't', *lda* must be at least max (1, *k*).

oa

INTEGER*1 for gemm_s8u8s32_compute

INTEGER*2 for gemm_s16s16s32_compute

Specifies the scalar offset value for the matrix *A*.

b INTEGER*1 for `gemm_s8u8s32_compute`

INTEGER*2 for `gemm_s16s16s32_compute`

transb = 'N' or 'n'	*transb* = 'T', 't'	*transb* = 'P' or 'p'
Array, size *ldb***n*. Before entry, the leading *k*-by-*n* part of the array *b* must contain the matrix *B*.	Array, size *ldb***k*. Before entry, the leading *n*-by-*k* part of the array *b* must contain the matrix *B*.	Array of size returned by `gemm_*_pack_get_size` and initialized using `gemm_*_pack`

ldb INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

If *transb* = 'N' or 'n', *ldb* must be at least `max (1, k)`.

If *transb* = 'T', 't', 'C', or 'c', *ldb* must be at least `max (1, n)`.

ob

INTEGER*1 for `gemm_s8u8s32_compute`

INTEGER*2 for `gemm_s16s16s32_compute`

Specifies the scalar offset value for the matrix *B*.

beta REAL

Specifies the scalar *beta*.

c INTEGER*4

Array, size *ldc* by *n*. Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is equal to zero, in which case *c* need not be set on entry.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

The value of *ldc* must be at least `max (1, m)`.

oc INTEGER*4

Array, size `len`. Specifies the scalar offset value for the matrix *C*.

If `offsetc = 'F' or 'f'`, `len` must be at least 1.

If `offsetc = 'C' or 'c'`, `len` must be at least `max(1, m)`.

If `offsetc = 'R' or 'r'`, `len` must be at least `max(1, n)`.

Output Parameters

c INTEGER*4

Overwritten by the matrix *alpha**(`op(A) + A_offset)*(op(B) + B_offset) + beta*C + C_offset`.

Example

See the following examples in the MKL installation directory to understand the use of these routines:

gemm_s8u8s32_compute: examples\blas\source\gemm_s8u8s32_computex.f

gemm_s16s16s32_compute: examples\blas\source\gemm_s16s16s32_computex.f

Application Notes

You can expand the matrix-matrix product in this manner:

`(op(A) + A_offset)*(op(B) + B_offset) = op(A)*op(B) + op(A)*B_offset + A_offset*op(B) + A_offset*B_offset`

After computing these four multiplication terms separately, they are summed from left to right. The results from the matrix-matrix product and the *C* matrix are scaled with *alpha* and *beta* floating-point values respectively using double-precision arithmetic. Before storing the results to the output *c* array, the floating-point values are rounded to the nearest integers.

In the event of overflow or underflow, the results depend on the architecture. The results are either unsaturated (wrapped) or saturated to maximum or minimum representable integer values for the data type of the output matrix.

See Also

`gemm_*_pack_get_size`
to return the number of bytes needed to store the packed matrix.

`gemm_*_pack`
to pack the matrix into the buffer allocated previously.

?gemm_free

Frees the storage previously allocated for the packed matrix (deprecated).

Syntax

```
call sgemm_free (dest)
call dgemm_free (dest)
```

Include Files

- `mkl.fi`

Description

The `?gemm_free` routine is one of a set of related routines that enable use of an internal packed storage. Call the `?gemm_free` routine last to release storage for the packed matrix structure allocated with `?gemm_alloc` (deprecated).

Input Parameters

dest	POINTER. Previously allocated storage.

Output Parameters

dest	The freed buffer.

See Also

`?gemm_pack` Performs scaling and packing of the matrix into the previously allocated buffer.

`?gemm_compute` Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.

`?gemm`
for a detailed description of general matrix multiplication.

gemm_*

Computes a matrix-matrix product with general integer matrices.

Syntax

`call gemm_s8u8s32(transa, transb, offsetc, m, n, k, alpha, a, lda, oa, b, ldb, ob, beta, c, ldc, oc)`

`call gemm_s16s16s32(transa, transb, offsetc, m, n, k, alpha, a, lda, oa, b, ldb, ob, beta, c, ldc, oc)`

Include Files

- `mkl.fi`

Description

The `gemm_*` routines compute a scalar-matrix-matrix product and adds the result to a scalar-matrix product. To get the final result, a vector is added to each row or column of the output matrix. The operation is defined as:

```
C := alpha*(op(A) + A_offset)*(op(B) + B_offset) + beta*C + C_offset
```

where :

`op(X)` is either `op(X) = X` or `op(X) = X^T`,

A_offset is an *m*-by-*k* matrix with every element equal to the value *oa*,

B_offset is a *k*-by-*n* matrix with every element equal to the value *ob*,

C_offset is an *m*-by-*n* matrix defined by the *oc* array as described in the description of the *offsetc* parameter,

alpha and *beta* are scalars,

A is a matrix such that `op(A)` is *m*-by-*k*,

B is a matrix such that `op(B)` is *k*-by-*n*,

and *C* is an *m*-by-*n* matrix.

Input Parameters

transa

CHARACTER*1. Specifies the form of op(*A*) used in the matrix multiplication:

if *transa* = `'N'` or `'n'`, then `op(A) = A`;

if *transa* = `'T'` or `'t'`, then `op(A) = A^T`.

transb

CHARACTER*1. Specifies the form of op(*B*) used in the matrix multiplication:

if *transb* = `'N'` or `'n'`, then `op(B) = B`;

if *transb* = `'T'` or `'t'`, then `op(B) = B^T`.

offsetc

CHARACTER*1. Specifies the form of *C_offset* used in the matrix multiplication.

offsetc = `'F'` or `'f'`: *oc* has a single element and every element of *C_offset* is equal to this element.

offsetc = 'C' or 'c': *oc* has a size of *m* and every column of *C_offset* is equal to *oc*.
offsetc = 'R' or 'r': *oc* has a size of *n* and every row of *C_offset* is equal to *oc*.

m — INTEGER. Specifies the number of rows of the matrix op(*A*) and of the matrix *C*. The value of *m* must be at least zero.

n — INTEGER. Specifies the number of columns of the matrix op(*B*) and the number of columns of the matrix *C*. The value of *n* must be at least zero.

k — INTEGER. Specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*). The value of *k* must be at least zero.

alpha — SINGLE PRECISION. Specifies the scalar *alpha*.

a — INTEGER*1 for gemm_s8u8s32.

INTEGER*2 for gemm_s16s16s32.

Array, size *lda* by *ka*, where *ka* is *k* when *transa* = 'N' or 'n', and is *m* otherwise. Before entry with *transa* = 'N' or 'n', the leading *m*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*m* part of the array *a* must contain the matrix *A*.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program.

When *transa* = 'N' or 'n', then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1, *k*).

oa — INTEGER*1 for gemm_s8u8s32.

INTEGER*2 for gemm_s16s16s32.

Specifies the scalar offset value for matrix *A*.

b — INTEGER*1 for gemm_s8u8s32. INTEGER*2 for gemm_s16s16s32. Array, size *ldb* by *kb*, where *kb* is *n* when *transa* = 'N' or 'n', and is *k* otherwise. Before entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *n*-by-*k* part of the array *b* must contain the matrix *B*.

ldb — INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

When *transb* = 'N' or 'n', then *ldb* must be at least max(1, *k*), otherwise *ldb* must be at least max(1, *n*).

ob — INTEGER*1 for gemm_s8u8s32. INTEGER*2 for gemm_s16s16s32. Specifies the scalar offset value for matrix *B*.

beta — SINGLE PRECISION. Specifies the scalar *beta*. When *beta* is equal to zero, then *c* need not be set on input.

c — INTEGER*4

Array, size *ldc* by *n*. Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is equal to zero, in which case *c* need not be set on entry.

ldc — INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

The value of *ldc* must be at least max(1, *m*).

oc — Array, size *len*. Specifies the offset values for matrix *C*.

If *offsetc* = 'F' or 'f': *len* must be at least 1.
If *offsetc* = 'C' or 'c': *len* must be at least max(1, *m*).
If *offsetc* = 'R' or 'r': *oc* must be at least max(1, *n*).

Output Parameters

c — Overwritten by *alpha**(op(*A*) + *A_offset*)*(op(*B*) + *B_offset*) + *beta***C*+ *C_offset*.

Application Notes

The matrix-matrix product can be expanded:

(op(*A*) + *A_offset*)*(op(*B*) + *B_offset*)

= op(*A*)*op(*B*) + op(*A*)**B_offset* + *A_offset**op(*B*) + *A_offset***B_offset*

After computing these four multiplication terms separately, they are summed from left to right. The results from the matrix-matrix product and the *C* matrix are scaled with *alpha* and *beta* floating-point values respectively using double-precision arithmetic. Before storing the results to the output *c* array, the floating-point values are rounded to the nearest integers. In the event of overflow or underflow, the results depend on the architecture . The results are either unsaturated (wrapped) or saturated to maximum or minimum representable integer values for the data type of the output matrix.

When using cblas_gemm_s8u8s32 with row-major layout, the data types of *A* and *B* must be swapped. That is, you must provide an 8-bit unsigned integer array for matrix *A* and an 8-bit signed integer array for matrix *B*.

Intermediate integer computations in gemm_s8u8s32 on 64-bit Intel® Advanced Vector Extensions 2 (Intel® AVX2) and Intel® Advanced Vector Extensions 512 (Intel® AVX-512) architectures without Vector Neural Network Instructions (VNNI) extensions can saturate. This is because only 16-bits are available for the accumulation of intermediate results. You can avoid integer saturation by maintaining all integer elements of A or B matrices under 8 bits.

?gemv_batch_strided

Computes groups of matrix-vector product with general matrices.

Syntax

```
call sgemv_batch_strided(trans, m, n, alpha, a, lda, stridea, x, incx, stridex, beta, y, incy, stridey, batch_size)
```

```
call dgemv_batch_strided(trans, m, n, alpha, a, lda, stridea, x, incx, stridex, beta, y, incy, stridey, batch_size)
```

```
call cgemv_batch_strided(trans, m, n, alpha, a, lda, stridea, x, incx, stridex, beta, y, incy, stridey, batch_size)
```

call zgemv_batch_strided(*trans*, *m*, *n*, *alpha*, *a*, *lda*, *stridea*, *x*, *incx*, *stridex*, *beta*, *y*, *incy*, *stridey*, *batch_size*)

Include Files

- mkl.fi

Description

The ?gemv_batch_strided routines perform a series of matrix-vector product added to a scaled vector. They are similar to the ?gemv routine counterparts, but the ?gemv_batch_strided routines perform matrix-vector operations with groups of matrices and vectors.

All matrices *a* and vectors *x* and *y* have the same parameters (size, increments) and are stored at constant *stridea*, *stridex*, and *stridey* from each other. The operation is defined as

```
for i = 0 … batch_size – 1
    A is a matrix at offset i * stridea in a
    X and Y are vectors at offset i * stridex and i * stridey in x and y
    Y = alpha * op(A) * X + beta * Y
end for
```

Input Parameters

trans

CHARACTER*1.

Specifies op(A) the transposition operation applied to the *A* matrices.

if *trans* = 'N' or 'n' , then op(A) = A;

if *trans* = 'T' or 't' , then op(A) = A';

if *trans* = 'C' or 'c' , then op(A) = conjg(A').

m

INTEGER. Number of rows of the matrices *A*. The value of *m* must be at least 0.

n

INTEGER. Number of columns of the matrices *A*. The value of *n* must be at least 0.

alpha

REAL for sgemv_batch_strided

DOUBLE PRECISION for dgemv_batch_strided

COMPLEX for cgemv_batch_strided

DOUBLE COMPLEX for zgemv_batch_strided

Specifies the scalar *alpha*.

a

REAL for sgemv_batch_strided

DOUBLE PRECISION for dgemv_batch_strided

COMPLEX for cgemv_batch_strided

DOUBLE COMPLEX for zgemv_batch_strided

Array holding all the input matrix *A*. Must be of size at least *lda***k* + *stridea* * (*batch_size* -1) where *k* is *n* if column major layout is used or *m* if row major layout is used.

lda

INTEGER. Specifies the leading dimension of the matrix*A*. It must be positive and at least *m* .

stridea	INTEGER. Stride between two consecutive *A* matrices. Must be at least 0.
x	REAL for sgemv_batch_strided DOUBLE PRECISION for dgemv_batch_strided COMPLEX for cgemv_batch_strided DOUBLE COMPLEX for zgemv_batch_strided Array holding all the input vector *x*. Must be of size at least (1 + (*len*-1)*abs(*incx*)) + *stridex* * (*batch_size* - 1) where *len* is *n* if the *A* matrix is not transposed or *m* otherwise.
incx	INTEGER. Stride between two consecutive elements of the *x* vectors. Must not be zero.
stridex	INTEGER. Stride between two consecutive *x* vectors, must be at least 0.
beta	REAL for sgemv_batch_strided DOUBLE PRECISION for dgemv_batch_strided COMPLEX for cgemv_batch_strided DOUBLE COMPLEX for zgemv_batch_strided Specifies the scalar *beta*.
y	REAL for sgemv_batch_strided DOUBLE PRECISION for dgemv_batch_strided COMPLEX for cgemv_batch_strided DOUBLE COMPLEX for zgemv_batch_strided Array holding all the input vectors *y*. Must be of size at least *batch_size* * *stridey*.
incy	INTEGER. Stride between two consecutive elements of the *y* vectors. Must not be zero.
stridey	INTEGER. Stride between two consecutive *y* vectors, must be at least (1 + (len-1)*abs(*incy*)) where len is *m* if the matrix *A* is non transpose or *n* otherwise.
batch_size	INTEGER. Number of gemv computations to perform and *a* matrices, *x* and *y* vectors. Must be at least 0.

Output Parameters

y	Array holding the *batch_size* updated vector *y*.

?gemv_batch

Computes groups of matrix-vector product with general matrices.

Syntax

call sgemv_batch(*trans_array*, *m_array*, *n_array*, *alpha_array*, *a_array*, *lda_array*, *x_array*, *incx_array*, *beta_array*, *y_array*, *incy_array*, *group_count*, *group_size*)

call dgemv_batch(*trans_array*, *m_array*, *n_array*, *alpha_array*, *a_array*, *lda_array*, *x_array*, *incx_array*, *beta_array*, *y_array*, *incy_array*, *group_count*, *group_size*)

call cgemv_batch(*trans_array*, *m_array*, *n_array*, *alpha_array*, *a_array*, *lda_array*, *x_array*, *incx_array*, *beta_array*, *y_array*, *incy_array*, *group_count*, *group_size*)

call zgemv_batch(*trans_array*, *m_array*, *n_array*, *alpha_array*, *a_array*, *lda_array*, *x_array*, *incx_array*, *beta_array*, *y_array*, *incy_array*, *group_count*, *group_size*)

Include Files

- mkl.fi

Description

The ?gemv_batch routines perform a series of matrix-vector product added to a scaled vector. They are similar to the ?gemv routine counterparts, but the ?gemv_batch routines perform matrix-vector operations with groups of matrices and vectors.

Each group contains matrices and vectors with the same parameters (size, increments). The operation is defined as:

```
idx = 0
For i = 0 … group_count – 1
    trans, m, n, alpha, lda, incx, beta, incy and group_size at position i in trans_array,
m_array, n_array, alpha_array, lda_array, incx_array, beta_array, incy_array and group_size_array
    for j = 0 … group_size – 1
        a is a matrix of size mxn at position idx in a_array
        x and y are vectors of size m or n depending on trans, at position idx in x_array and
y_array
        y := alpha * op(a) * x + beta * y
        idx := idx + 1
    end for
end for
```

The number of entries in *a_array*, *x_array*, and *y_array* is *total_batch_count* = the sum of all of the *group_size* entries.

Input Parameters

trans_array

CHARACTER*1.

Array of size *group_count*. For the group *i*, $trans_i$ = *trans_array*[i] specifies the transposition operation applied to *A*.

if *trans* = 'N' or 'n' , then op(A) = A;

if *trans* = 'T' or 't' , then op(A) = A';

if *trans* = 'C' or 'c' , then op(A) = conjg(A').

m_array

INTEGER. Array of size *group_count*. For the group *i*, m_i = *m_array*[i] is the number of rows of the matrix *A*.

n_array

INTEGER. Array of size *group_count*. For the group *i*, n_i = *n_array*[i] is the number of columns in the matrix *A*.

alpha_array

REAL for sgemv_batch

	DOUBLE PRECISION for dgemv_batch COMPLEX for cgemv_batch DOUBLE COMPLEX for zgemv_batch Array of size *group_count*. For the group *i*, $alpha_i$ = *alpha_array*[i] is the scalar *alpha*.
a_array	INTEGER*8 for Intel® 64 architecture INTEGER*4 for IA32 architecture Array of size *total_batch_count* of pointers used to store *A* matrices. The array allocated for the *A* matrices of the group *i* must be of size at least $lda_i * n_i$ if column major layout is used or at least $lda_i * m_i$ is row major layout is used.
lda_array	INTEGER. Array of size *group_count*. For the group *i*, lda_i = *lda_array*[i] is the leading dimension of the matrix *A*. It must be positive and at least m_i .
x_array	INTEGER*8 for Intel® 64 architecture INTEGER*4 for IA32 architecture Array of size *total_batch_count* of pointers used to store *x* vectors. The array allocated for the *x* vectors of the group *i* must be of size at least (1 + len_i – 1)*abs($incx_i$)) where len_i is n_i if the *A* matrix is not transposed or m_i otherwise.
incx_array	INTEGER. Array of size *group_count*. For the group *i*, $incx_i$ = *incx_array*[i] is the stride of vector *x*. Must not be zero.
beta_array	REAL for sgemv_batch DOUBLE PRECISION for dgemv_batch COMPLEX for cgemv_batch DOUBLE COMPLEX for zgemv_batch Array of size *group_count*. For the group *i*, $beta_i$ = *beta_array*[i] is the scalar *beta*.
y_array	INTEGER*8 for Intel® 64 architecture INTEGER*4 for IA32 architecture Array of size *total_batch_count* of pointers used to store *y* vectors. The array allocated for the *y* vectors of the group *i* must be of size at least (1 + len_i – 1)*abs($incy_i$)) where len_i is m_i if the *A* matrix is not transposed or n_i otherwise.
incy_array	INTEGER. Array of size *group_count*. For the group *i*, $incy_i$ = *incy_array*[i] is the stride of vector *y*. Must not be zero.
group_count	INTEGER. Number of groups. Must be at least 0.
group_size	INTEGER.

Array of size *group_count*. The element *group_count*[i] is the number of operations in the group *i*. Each element in *group_count* must be at least 0.

Output Parameters

y_array — Array of pointers holding the *total_batch_count* updated vector *y*.

?dgmm_batch_strided

Computes groups of matrix-vector product using general matrices.

Syntax

```
call sdgmm_batch_strided(left_right, m, n, a, lda, stridea, x, incx, stridex, c, ldc, stridec, batch_size)
call ddgmm_batch_strided(left_right, m, n, a, lda, stridea, x, incx, stridex, c, ldc, stridec, batch_size)
call cdgmm_batch_strided(left_right, m, n, a, lda, stridea, x, incx, stridex, c, ldc, stridec, batch_size)
call zdgmm_batch_strided(left_right, m, n, a, lda, stridea, x, incx, stridex, c, ldc, stridec, batch_size)
```

Include Files

- mkl.fi

Description

The ?dgmm_batch_strided routines perform a series of diagonal matrix-matrix product. The diagonal matrices are stored as dense vectors and the operations are performed with group of matrices and vectors.

All matrices *a* and *c* and vector *x* have the same parameters (size, increments) and are stored at constant stride, respectively, given by *stridea*, *stridec*, and *stridex* from each other. The operation is defined as

```
for i = 0 … batch_size – 1
    A and C are matrices at offset i * stridea in a and i * stridec in c
    X is a vector at offset i * stridex in x
    C = diag(X) * A or C = A * diag(X)
end for
```

Input Parameters

left_right — CHARACTER*1.

Specifies the position of the diagonal matrix in the matrix product

if *left_right* = 'L' or 'l' , then C = diag(X) * A;

if *left_right* = 'R' or 'r' , then C = A * diag(X).

m — INTEGER. Number of rows of the matrices A and C. The value of *m* must be at least 0.

n — INTEGER. Number of columns of the matrices A and C. The value of *n* must be at least 0.

a — REAL for sdgmm_batch_strided

	DOUBLE PRECISION for ddgmm_batch_strided COMPLEX for cdgmm_batch_strided DOUBLE COMPLEX for zdgmm_batch_strided Array holding all the input matrix *A*. Must be of size at least *lda***k* + *stridea* * (*batch_size* -1) where *k* is *n* if column major layout is used or *m* if row major layout is used.
lda	INTEGER. Specifies the leading dimension of the matrix*A*. It must be positive and at least *m* .
stridea	INTEGER. Stride between two consecutive *A* matrices, must be at least 0.
x	REAL for sdgmm_batch_strided DOUBLE PRECISION for ddgmm_batch_strided COMPLEX for cdgmm_batch_strided DOUBLE COMPLEX for zdgmm_batch_strided Array holding all the input vector *x*. Must be of size at least (1 + (*len* -1)*abs(*incx*)) + *stridex* * (*batch_size* - 1) where *len* is *n* if the diagonal matrix is on the right of the product or *m* otherwise.
incx	INTEGER. Stride between two consecutive elements of the *x* vectors.
stridex	INTEGER. Stride between two consecutive *x* vectors, must be at least 0.
c	REAL for sdgmm_batch_strided DOUBLE PRECISION for ddgmm_batch_strided COMPLEX for cdgmm_batch_strided DOUBLE COMPLEX for zdgmm_batch_strided Array holding all the input matrix *C*. Must be of size at least *batch_size* * *stridec*.
ldc	INTEGER. Specifies the leading dimension of the matrix *C*. It must be positive and at least *m* .
stridec	INTEGER. Stride between two consecutive *A* matrices, must be at least *ldc* * *n* .
batch_size	INTEGER. Number of dgmm computations to perform and a *c* matrices and *x* vectors. Must be at least 0.

Output Parameters

c	Array holding the *batch_size* updated matrices *c*.

?dgmm_batch

Computes groups of matrix-vector product using general matrices.

Syntax

`call sdgmm_batch(left_right_array, m_array, n_array, a_array, lda_array, x_array, incx_array, c_array, ldc_array, group_count, group_size)`

`call ddgmm_batch(left_right_array, m_array, n_array, a_array, lda_array, x_array, incx_array, c_array, ldc_array, group_count, group_size)`

`call cdgmm_batch(left_right_array, m_array, n_array, a_array, lda_array, x_array, incx_array, c_array, ldc_array, group_count, group_size)`

`call zdgmm_batch(left_right_array, m_array, n_array, a_array, lda_array, x_array, incx_array, c_array, ldc_array, group_count, group_size)`

Include Files

- `mkl.fi`

Description

The `?dgmm_batch` routines perform a series of diagonal matrix-matrix product. The diagonal matrices are stored as dense vectors and the operations are performed with group of matrices and vectors. .

Each group contains matrices and vectors with the same parameters (size, increments). The operation is defined as:

```
idx = 0
For i = 0 … group_count – 1
    left_right, m, n, lda, incx, ldc and group_size at position i in left_right_array, m_array,
n_array, lda_array, incx_array, ldc_array and group_size_array
    for j = 0 … group_size – 1
        a and c are matrices of size mxn at position idx in a_array and c_array
        x is a vector of size m or n depending on left_right, at position idx in x_array
        if (left_right == oneapi::mkl::side::left) c := diag(x) * a
        else c := a * diag(x)
        idx := idx + 1
    end for
end for
```

The number of entries in *a_array*, *x_array*, and *c_array* is *total_batch_count* = the sum of all of the *group_size* entries.

Input Parameters

left_right_array

`CHARACTER*1`.

Array of size *group_count*. For the group *i*, $left\_right_i$ = *left_right_array*[i] specifies the position of the diagonal matrix in the matrix product.

if $left\_right_i$ = 'L' or 'l' , then *C* = diag(X) * *A*.

if $left\_right_i$ = 'R' or 'r' , then *C* = *A* * diag(X).

m_array

`INTEGER`. Array of size *group_count*. For the group *i*, m_i = *m_array*[i] is the number of rows of the matrix *A* and *C*.

n_array — INTEGER. Array of size *group_count*. For the group *i*, n_i = *n_array*[i] is the number of columns in the matrix *A* and *C*.

a_array — INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA32 architecture

Array of size *total_batch_count* of pointers used to store *A* matrices. The array allocated for the *A* matrices of the group *i* must be of size at least $lda_i * n_i$.

lda_array — INTEGER. Array of size *group_count*. For the group *i*, lda_i = *lda_array*[i] is the leading dimension of the matrix *A*. It must be positive and at least m_i .

x_array — INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA32 architecture

Array of size *total_batch_count* of pointers used to store *x* vectors. The array allocated for the *x* vectors of the group *i* must be of size at least (1 + len_i – 1)*abs($incx_i$)) where len_i is n_i if the diagonal matrix is on the right of the product or m_i otherwise.

incx_array — INTEGER. Array of size *group_count*. For the group *i*, $incx_i$ = *incx_array*[i] is the stride of vector *x*.

c_array — INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA32 architecture

Array of size *total_batch_count* of pointers used to store *C* matrices. The array allocated for the *C* matrices of the group *i* must be of size at least $ldc_i * n_i$,

ldc_array — INTEGER.

Array of size *group_count*. For the group *i*, ldc_i = *ldc_array*[i] is the leading dimension of the matrix *C*. It must be positive and at least m_i .

group_count — INTEGER.

Number of groups. Must be at least 0.

group_size — INTEGER.

Array of size *group_count*. The element *group_count*[i] is the number of operations in the group *i*. Each element in *group_size* must be at least 0.

Output Parameters

c_array — Array of pointers holding the *total_batch_count* updated matrix *C*.

mkl_jit_create_?gemm

Create a GEMM kernel that computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product.

Syntax

```
status = mkl_jit_create_sgemm(jitter, transa, transb, m, n, k, alpha, lda, ldb, beta, ldc)
```

```
status = mkl_jit_create_dgemm(jitter, transa, transb, m, n, k, alpha, lda, ldb, beta, ldc)
status = mkl_jit_create_cgemm(jitter, transa, transb, m, n, k, alpha, lda, ldb, beta, ldc)
status = mkl_jit_create_zgemm(jitter, transa, transb, m, n, k, alpha, lda, ldb, beta, ldc)
```

Include Files

- mkl_blas.f90

Description

The mkl_jit_create_?gemm functions belong to a set of related routines that enable use of just-in-time code generation.

The mkl_jit_create_?gemm functions create a handle to a just-in-time code generator (a jitter) and generate a GEMM kernel that computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product, with general matrices. The operation of the generated GEMM kernel is defined as follows:

```
C := alpha*op(A)*op(B) + beta*C
```

Where:

- op(X) is either op(X) = X or op(X) = X^T or op(X) = X^H
- alpha and beta are scalars
- A, B, and C are matrices
- op(A) is an m-by-k matrix
- op(B) is a k-by-n matrix
- C is an m-by-n matrix

NOTE
Generating a new kernel with mkl_jit_create_?gemm involves moderate runtime overhead. To benefit from JIT code generation, use this feature when you need to call the generated kernel many times (for example, several hundred calls).

NOTE
The JIT API requires Fortran 90 and the ISO_C_BINDING module.

Input Parameters

transa

CHARACTER*1.

Specifies the form of op(A) used in the generated matrix multiplication:

- if *transa* = 'N', then op(A) = A
- if *transa* = 'T', then op(A) = A^T
- if *transa* = 'C', then op(A) = A^H

transb

CHARACTER*1.

Specifies the form of op(B) used in the generated matrix multiplication:

- if *transb* = 'N', then op(B) = B

- if *transb* = 'T', then op(B) = B^T
- if *transb* = 'C', then op(B) = B^H

m

INTEGER.

Specifies the number of rows of the matrix op(A) and of the matrix C. The value of *m* must be at least zero.

n

INTEGER.

Specifies the number of columns of the matrix op(B) and of the matrix C. The value of *n* must be at least zero.

k

INTEGER.

Specifies the number of columns of the matrix op(A) and the number of rows of the matrix op(B). The value of *k* must be at least zero.

alpha

REAL for mkl_jit_create_sgemm

DOUBLE PRECISION for mkl_jit_create_dgemm.

COMPLEX for mkl_jit_create_cgemm

DOUBLE COMPLEX for mkl_jit_create_zgemm

Specifies the scalar *alpha*.

lda

INTEGER.

Specifies the leading dimension of *a*.

- If *transa* = 'N' *lda* must be at least max(1,m).
- If *transa* = 'T' or *transa* = 'C', *lda* must be at least max(1,k).

ldb

INTEGER.

Specifies the leading dimension of *b*:

- If *transb* = 'N' *ldb* must be at least max(1,k).
- If *transb* = 'T' or *transb* = 'C', *ldb* must be at least max(1,n).

beta

REAL for mkl_jit_create_sgemm

DOUBLE PRECISION for mkl_jit_create_dgemm.

COMPLEX for mkl_jit_create_cgemm

DOUBLE COMPLEX for mkl_jit_create_zgemm

Specifies the scalar *beta*.

ldc

INTEGER.

Specifies the leading dimension of *c* which must be at least max(1,m).

Output Parameters

jitter

TYPE(C_PTR). C pointer to a handle to the newly created code generator.

Return Values

`status`	`INTEGER` Returns one of the following: • `MKL_JIT_ERROR` if the handle cannot be created (no memory) *—or—* • `MKL_JIT_SUCCESS` if the jitter has been created and the GEMM kernel was successfully created *—or—* • `MKL_NO_JIT` if the jitter has been created, but a JIT GEMM kernel was not created because JIT is not beneficial for the given input parameters. The function pointer returned by `mkl_jit_get_?gemm_ptr` will call standard (non-JIT) GEMM.

mkl_jit_get_?gemm_ptr

Return the GEMM kernel associated with a jitter previously created with `mkl_jit_create_?gemm`*.*

Syntax

c_func = mkl_jit_get_sgemm_ptr(*jitter*)

c_func = mkl_jit_get_dgemm_ptr(*jitter*)

c_func = mkl_jit_get_cgemm_ptr(*jitter*)

c_func = mkl_jit_get_zgemm_ptr(*jitter*)

Include Files

- `mkl_blas.f90`

Description

The `mkl_jit_get_?gemm_ptr` functions belong to a set of related routines that enable use of just-in-time code generation.

The `mkl_jit_get_?gemm_ptr` functions take as input a jitter previously created with `mkl_jit_create_?gemm`, and return the GEMM kernel associated with that jitter. The returned GEMM kernel computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product, with general matrices. The operation is defined as follows:

```
C := alpha*op(A)*op(B) + beta*C
```

Where:

- `op(X)` is one of `op(X) = X` or `op(X) = X`^{T} or `op(X) = X`^{H}
- `alpha` and `beta` are scalars
- `A`, `B`, and `C` are matrices
- `op(A)` is an `m`-by-`k` matrix
- `op(B)` is a `k`-by-`n` matrix
- C is an `m`-by-`n` matrix

NOTE
Generating a new kernel with `mkl_jit_create_?gemm` involves moderate runtime overhead. To benefit from JIT code generation, use this feature when you need to call the generated kernel many times (for example, several hundred calls).

NOTE
The JIT API requires Fortran 90 and the ISO_C_BINDING module.

Input Parameter

jitter

`TYPE(C_PTR), VALUE`

Handle to the code generator.

Return Values

c_func

`TYPE(C_FUNPTR)`

If the jitter input is not a C NULL pointer, returns a C function pointer to a GEMM kernel. The returned C function pointer must be converted to a Fortran procedure pointer (of abstract interface `?gemm_jit_kernel_t`) using `C_F_PROCPOINTER`. The GEMM kernel can then be called with four parameters: the jitter and the three matrices *a*, b, and *c*. Otherwise, returns a C NULL pointer.

If *transa*, *transb*, *m*, *n*, *k*, *lda*, *ldb*, and *ldc* are the parameters used during the creation of the input jitter, then:

a

transa = 'N'	*transa* = 'T' or *transa* = 'C'
Array of size *lda***k* Before calling the returned function pointer, the leading *m*-by-*k* part of the array *a* must contain the matrix A.	Array of size *lda***m* Before calling the returned function pointer, the leading *k*-by-*m* part of the array *a* must contain the matrix A.

b

transb = 'N'	*transb* = 'T' or *transb* = 'C'
Array of size *ldb***n* Before calling the returned function pointer, the leading *k*-by-*n* part of the array *b* must contain the matrix B.	Array of size *ldb***k* Before calling the returned function pointer, the leading *n*-by-*k* part of the array *b* must contain the matrix B.

c

Array of size *ldc***n*

Before calling the returned function pointer, the leading *m*-by-*n* part of the array *c* must contain the matrix C.

mkl_jit_destroy

Delete the jitter previously created with `mkl_jit_create_?gemm` *as well as the GEMM kernel that it contains.*

Syntax

`status = mkl_jit_destroy (jitter)`

Include Files

- `mkl_blas.f90`

Description

The `mkl_jit_destroy` function belongs to a set of related routines that enable use of just-in-time code generation.

The `mkl_jit_destroy` function takes as input a jitter previously created with `mkl_jit_create_?gemm` and deletes the jitter as well as the GEMM kernel that it contains.

NOTE
Generating a new kernel with `mkl_jit_create_?gemm` involves moderate runtime overhead. To benefit from JIT code generation, use this feature when you need to call the generated kernel many times (for example, several hundred calls).

NOTE
The JIT API requires Fortran 90 and the ISO_C_BINDING module.

Input Parameter

`jitter`	`TYPE(C_PTR), VALUE` Jitter handle

Return Values

`status`	`INTEGER` Returns one of the following: • `MKL_JIT_ERROR` if the pointer is not NULL and is not a handle on a jitter—that is, if it was not created with `mkl_jit_create_?gemm` *—or—* • `MKL_JIT_SUCCESS` if the jitter has been successfully destroyed

LAPACK Routines

Intel® oneAPI Math Kernel Library (oneMKL)implements routines from the LAPACK package that are used for solving systems of linear equations, linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. The library includes LAPACK routines for both real and complex data. Routines are supported for systems of equations with the following types of matrices:

- General
- Banded
- Symmetric or Hermitian positive-definite (full, packed, and rectangular full packed (RFP) storage)
- Symmetric or Hermitian positive-definite banded
- Symmetric or Hermitian indefinite (both full and packed storage)
- Symmetric or Hermitian indefinite banded
- Triangular (full, packed, and RFP storage)
- Triangular banded
- Tridiagonal
- Diagonally dominant tridiagonal.

NOTE
Different arrays used as parameters to Intel® MKL LAPACK routines must not overlap.

Warning
LAPACK routines assume that input matrices do not contain IEEE 754 special values such as `INF` or `NaN` values. Using these special values may cause LAPACK to return unexpected results or become unstable.

Intel MKL supports the Fortran 95 interface, which uses simplified routine calls with shorter argument lists, in addition to the FORTRAN 77 interface to LAPACK computational and driver routines. The syntax section of the routine description gives the calling sequence for the Fortran 95 interface, where available, immediately after the FORTRAN 77 calls.

Naming Conventions for LAPACK Routines

To call one of the routines from a FORTRAN 77 program, you can use the LAPACK name.

LAPACK names have the structure `?yyzzz` or `?yyzz`, where the initial symbol `?` indicates the data type:

`s`	real, single precision
`c`	complex, single precision
`d`	real, double precision
`z`	complex, double precision

Some routines can have combined character codes, such as `ds` or `zc`.

The Fortran 95 interfaces to the LAPACK computational and driver routines are the same as the FORTRAN 77 names but without the first letter that indicates the data type. For example, the name of the routine that performs a triangular factorization of general real matrices in Fortran 95 is `getrf`. Different data types are handled through the definition of a specific internal parameter that refers to a module block with named constants for single and double precision.

Fortran 95 Interface Conventions for LAPACK Routines

Intel® oneAPI Math Kernel Library (oneMKL) implements the Fortran 95 interface to LAPACK through wrappers that call respective FORTRAN 77 routines. This interface uses such Fortran 95 features as assumed-shape arrays and optional arguments to provide simplified calls to LAPACK routines with fewer arguments.

NOTE
For LAPACK, Intel® oneAPI Math Kernel Library (oneMKL) offers two types of the Fortran 95 interfaces:

- using `mkl_lapack.fi` only through the `include 'mkl_lapack.fi'` statement. Such interfaces allow you to make use of the original LAPACK routines with all their arguments
- using `lapack.f90` that includes improved interfaces. This file is used to generate the module files `lapack95.mod` and `f95_precision.mod`. See also the section "Fortran 95 interfaces and wrappers to LAPACK and BLAS" of the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details. The module files are used to process the FORTRAN use clauses referencing the LAPACK interface: `use lapack95` and `use f95_precision`.

The main conventions for the Fortran 95 interface are as follows:

- The names of arguments used in Fortran 95 call are typically the same as for the respective generic (FORTRAN 77) interface. In rare cases, formal argument names may be different. For instance, *select* instead of *selctg*.
- Input arguments such as array dimensions are not required in Fortran 95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.

  Another type of generic arguments that are skipped in the Fortran 95 interface are arguments that represent workspace arrays (such as *work*, *rwork*, and so on). The only exception are cases when workspace arrays return significant information on output.

  NOTE
  Internally, workspace arrays are allocated by the Fortran 95 interface wrapper, and are of optimal size for the best performance of the routine.

  An argument can also be skipped if its value is completely defined by the presence or absence of another argument in the calling sequence, and the restored value is the only meaningful value for the skipped argument.
- Some generic arguments are declared as optional in the Fortran 95 interface and may or may not be present in the calling sequence. An argument can be declared optional if it meets one of the following conditions:
  - If an argument value is completely defined by the presence or absence of another argument in the calling sequence, it can be declared optional. The difference from the skipped argument in this case is that the optional argument can have some meaningful values that are distinct from the value reconstructed by default. For example, if some argument (like *jobz*) can take only two values and one of these values directly implies the use of another argument, then the value of *jobz* can be uniquely reconstructed from the actual presence or absence of this second argument, and *jobz* can be omitted.
  - If an input argument can take only a few possible values, it can be declared as optional. The default value of such argument is typically set as the first value in the list and all exceptions to this rule are explicitly stated in the routine description.
  - If an input argument has a natural default value, it can be declared as optional. The default value of such optional argument is set to its natural default value.
- Argument *info* is declared as optional in the Fortran 95 interface. If it is present in the calling sequence, the value assigned to *info* is interpreted as follows:
  - If this value is more than -1000, its meaning is the same as in the FORTRAN 77 routine.
  - If this value is equal to -1000, it means that there is not enough work memory.
  - If this value is equal to -1001, incompatible arguments are present in the calling sequence.
  - If this value is equal to *-i*, the *i*th parameter (counting parameters in the FORTRAN 77 interface, not the Fortran 95 interface) had an illegal value.
- Optional arguments are given in square brackets in the Fortran 95 call syntax.

The "Fortran 95 Notes" subsection at the end of the topic describing each routine details concrete rules for reconstructing the values of the omitted optional parameters.

Intel® MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib Implementation

The following list presents general digressions of the Intel® oneAPI Math Kernel Library (oneMKL) LAPACK95 implementation from the Netlib analog:

- The Intel® oneAPI Math Kernel Library (oneMKL) Fortran 95 interfaces are provided for pure procedures.
- Names of interfaces do not contain the `LA_` prefix.
- An optional array argument always has the `target` attribute.
- Functionality of the Intel® oneAPI Math Kernel Library (oneMKL) LAPACK95 wrapper is close to the FORTRAN 77 original implementation in the`getrf`, `gbtrf`, and `potrf` interfaces.
- If *jobz* argument value specifies presence or absence of *z* argument, then *z* is always declared as optional and *jobz* is restored depending on whether *z* is present or not.
- To avoid double error checking, processing of the *info* argument is limited to checking of the allocated memory and disarranging of optional arguments.
- If an argument that is present in the list of arguments completely defines another argument, the latter is always declared as optional.

You can transform an application that uses the Netlib LAPACK interfaces to ensure its work with the Intel® oneAPI Math Kernel Library (oneMKL) interfaces providing that:

a. The application is correct, that is, unambiguous, compiler-independent, and contains no errors.
b. Each routine name denotes only one specific routine. If any routine name in the application coincides with a name of the original Netlib routine (for example, after removing the `LA_` prefix) but denotes a routine different from the Netlib original routine, this name should be modified through context name replacement.

You should transform your application in the following cases:

- When using the Netlib routines that differ from the Intel® oneAPI Math Kernel Library (oneMKL) routines only by the`LA_` prefix or in the array attribute `target`. The only transformation required in this case is context name replacement.
- When using Netlib routines that differ from the Intel® oneAPI Math Kernel Library (oneMKL) routines by the`LA_` prefix, the `target` array attribute, and the names of formal arguments. In the case of positional passing of arguments, no additional transformation except context name replacement is required. In the case of the keywords passing of arguments, in addition to the context name replacement the names of mismatching keywords should also be modified.
- When using the Netlib routines that differ from the respective Intel® oneAPI Math Kernel Library (oneMKL) routines by the`LA_` prefix, the `target`array attribute, sequence of the arguments, arguments missing in Intel® oneAPI Math Kernel Library (oneMKL) but present in Netlib and, vice versa, present in Intel® oneAPI Math Kernel Library (oneMKL) but missing in Netlib. Remove the differences in the sequence and range of the arguments in process of all the transformations when you use the Netlib routines specified by this bullet and the preceding bullet.
- When using the `getrf`, `gbtrf`, and `potrf`interfaces, that is, new functionality implemented in Intel® oneAPI Math Kernel Library (oneMKL) but unavailable in the Netlib source. To override the differences, build the desired functionality explicitly with the Intel® oneAPI Math Kernel Library (oneMKL) means or create a new subroutine with the new functionality, using specific MKL interfaces corresponding to LAPACK 77 routines. You can call the LAPACK 77 routines directly but using the new Intel® oneAPI Math Kernel Library (oneMKL) interfaces is preferable. Note that if the transformed application calls`getrf`, `gbtrf` or `potrf` without controlling arguments *rcond* and *norm*, just context name replacement is enough in modifying the calls into the Intel® oneAPI Math Kernel Library (oneMKL) interfaces, as described in the first bullet above. The Netlib functionality is preserved in such cases.
- When using the Netlib auxiliary routines. In this case, call a corresponding subroutine directly, using the Intel® oneAPI Math Kernel Library (oneMKL) LAPACK 77 interfaces.

Transform your application as follows:

1. Make sure conditions a. and b. are met.
2. Select Netlib LAPACK 95 calls. For each call, do the following:
   - Select the type of digression and do the required transformations.
   - Revise results to eliminate unneeded code or data, which may appear after several identical calls.
3. Make sure the transformations are correct and complete.

Matrix Storage Schemes for LAPACK Routines

LAPACK routines use the following matrix storage schemes:

- *Full storage*: an *m*-by-*n* matrix *A* is stored in a two-dimensional array `a`, with the matrix element a_{ij} (i = 1..*mj* = 1..*n*), and stored in the array element `a(i,j)`.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: an *m*-by-*n* band matrix with *kl* sub-diagonals and *ku* superdiagonals is stored compactly in a two-dimensional array *ab* with `kl+ku+1` rows and *n* columns. Columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.
- *Rectangular Full Packed (RFP) storage*: the upper or lower triangle of the matrix is packed combining the full and packed storage schemes. This combination enables using half of the full storage as packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels as the full storage.

Generally in LAPACK routines, arrays that hold matrices in packed storage have names ending in *p*; arrays with matrices in band storage have names ending in *b*; arrays with matrices in the RFP storage have names ending in *fp*.

For more information on matrix storage schemes, see "Matrix Arguments" in "Routine and Function Arguments".

Mathematical Notation for LAPACK Routines

Descriptions of LAPACK routines use the following notation:

A^H	For an *M*-by-*N* matrix A, denotes the conjugate transposed *N*-by-*M* matrix with elements: $a^H_{ij} = \overline{a_{ji}}$ For a real-valued matrix, $A^H = A^T$.
$x \cdot y$	The *dot product* of two vectors, defined as: $x \cdot y = \sum_i x_i \overline{y_i}$
$Ax = b$	A system of linear equations with an *n*-by-*n* matrix $A = \{a_{ij}\}$, a right-hand side vector $b = \{b_i\}$, and an unknown vector $x = \{x_i\}$.
$AX = B$	A set of systems with a common matrix *A* and multiple right-hand sides. The columns of *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.
$\|x\|$	the vector with elements $\|x_i\|$ (absolute values of x_i).
$\|A\|$	the matrix with elements $\|a_{ij}\|$ (absolute values of a_{ij}).
$\|\|x\|\|_\infty = \max_i \|x_i\|$	The *infinity-norm* of the vector *x*.
$\|\|A\|\|_\infty = \max_i \Sigma_j \|a_{ij}\|$	The *infinity-norm* of the matrix *A*.
$\|\|A\|\|_1 = \max_j \Sigma_i \|a_{ij}\|$	The *one-norm* of the matrix *A*. $\|\|A\|\|_1 = \|\|A^T\|\|_\infty = \|\|A^H\|\|_\infty$
$\|\|x\|\|_2$	The *2-norm* of the vector *x*: $\|\|x\|\|_2 = (\Sigma_i \|x_i\|^2)^{1/2} = \|\|x\|\|_E$ (see the definition for *Euclidean norm* in this topic).
$\|\|A\|\|_2$	The *2-norm* (or *spectral norm*) of the matrix *A*.

$$\|A\|_2 = \max_i \sigma_i, \|A\|_2^2 = \max_{\|x\|_2=1} (Ax \cdot Ax)$$

$\|\|A\|\|_E$	The *Euclidean norm* of the matrix *A*: $\|\|A\|\|_E^2 = \Sigma_i \Sigma_j \|a_{ij}\|^2$.
$\kappa(A) = \|\|A\|\| \cdot \|\|A^{-1}\|\|$	The *condition number* of the matrix *A*.
λ_i	*Eigenvalues* of the matrix *A* (for the definition of eigenvalues, see Eigenvalue Problems).
σ_i	*Singular values* of the matrix *A*. They are equal to square roots of the eigenvalues of $A^H A$. (For more information, see Singular Value Decomposition).

Error Analysis

In practice, most computations are performed with rounding errors. Besides, you often need to solve a system $Ax = b$, where the data (the elements of *A* and *b*) are not known exactly. Therefore, it is important to understand how the data errors and rounding errors can affect the solution *x*.

Data perturbations. If *x* is the exact solution of $Ax = b$, and $x + \delta x$ is the exact solution of a perturbed problem $(A + \delta A)(x + \delta x) = (b + \delta b)$, then this estimate, given up to linear terms of perturbations, holds:

$$\frac{\|\delta x\|}{\|x\|} \le \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right)$$

where $A + \delta A$ is nonsingular and

$$\kappa(A) = \|A\| \|A^{-1}\|.$$

In other words, relative errors in *A* or *b* may be amplified in the solution vector *x* by a factor $\kappa(A) = \|\|A\|\| \|\|A^{-1}\|\|$ called the *condition number* of *A*.

Rounding errors have the same effect as relative perturbations $c(n)\varepsilon$ in the original data. Here ε is the *machine precision*, defined as the smallest positive number *x* such that $1 + x > 1$; and $c(n)$ is a modest function of the matrix order *n*. The corresponding solution error is

$\|\|\delta x\|\| / \|\|x\|\| \le c(n)\kappa(A)\varepsilon$. (The value of $c(n)$ is seldom greater than $10n$.)

NOTE
Machine precision depends on the data type used.

Thus, if your matrix *A* is *ill-conditioned* (that is, its condition number $\kappa(A)$ is very large), then the error in the solution *x* can also be large; you might even encounter a complete loss of precision. LAPACK provides routines that allow you to estimate $\kappa(A)$ (see Routines for Estimating the Condition Number) and also give you a more precise estimate for the actual solution error (see Refining the Solution and Estimating Its Error).

LAPACK Linear Equation Routines

This section describes routines for performing the following computations:

- factoring the matrix (except for triangular matrices)
- equilibrating the matrix (except for RFP matrices)
- solving a system of linear equations
- estimating the condition number of a matrix (except for RFP matrices)
- refining the solution of linear equations and computing its error bounds (except for RFP matrices)
- inverting the matrix.

To solve a particular problem, you can call two or more computational routines or call a corresponding driver routine that combines several tasks in one call. For example, to solve a system of linear equations with a general matrix, call `?getrf` (*LU* factorization) and then `?getrs` (computing the solution). Then, call `?gerfs` to refine the solution and get the error bounds. Alternatively, use the driver routine `?gesvx` that performs all these tasks in one call.

LAPACK Linear Equation Computational Routines

Table "Computational Routines for Systems of Equations with Real Matrices" lists the LAPACK computational routines (FORTRAN 77 and Fortran 95 interfaces) for factorizing, equilibrating, and inverting *real* matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error. Table "Computational Routines for Systems of Equations with Complex Matrices" lists similar routines for *complex* matrices. Respective routine names in the Fortran 95 interface are without the first symbol (see Routine Naming Conventions).

Computational Routines for Systems of Equations with Real Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	`?getrf`	`?geequ`, `?geequb`	`?getrs`	`?gecon`	`?gerfs`, `?gerfsx`	`?getri`
general band	`?gbtrf`	`?gbequ`, `?gbequb`	`?gbtrs`	`?gbcon`	`?gbrfs`, `?gbrfsx`	
general tridiagonal	`?gttrf`		`?gttrs`	`?gtcon`	`?gtrfs`	
diagonally dominant tridiagonal	`?dttrfb`		`?dttrsb`			
symmetric positive-definite	`?potrf`	`?poequ`, `?poequb`	`?potrs`	`?pocon`	`?porfs`, `?porfsx`	`?potri`
symmetric positive-definite, packed storage	`?pptrf`	`?ppequ`	`?pptrs`	`?ppcon`	`?pprfs`	`?pptri`
symmetric positive-definite, RFP storage	`?pftrf`		`?pftrs`			`?pftri`

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
symmetric positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
symmetric positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
symmetric indefinite	?sytrf	?syequb	?sytrs	?sycon	?syrfs,	?sytri
	?sytrf_rook		?sytrs_rook	?sycon_rook	?syrfsx	?sytri_rook
	?sytrf_rk		?sytrs2	?sycon_3		?sytri2
	?sytrf_aa		?sytrs3			?sytri2x
			?sytrs_aa			?sytri_3
symmetric indefinite, packed storage	?sptrf		?sptrs	?spcon	?sprfs	?sptri
	mkl_?spffrt2, mkl_?spffrtx					
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprfs	?tptri
triangular, RFP storage						?tftri
triangular band			?tbtrs	?tbcon	?tbrfs	

In the table above, ? denotes s (single precision) or d (double precision) for the FORTRAN 77 interface.

Computational Routines for Systems of Equations with Complex Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	?getrf	?geequ,	?getrs	?gecon	?gerfs,	?getri
		?geequb			?gerfsx	
general band	?gbtrf	?gbequ,	?gbtrs	?gbcon	?gbrfs,	
		?gbequb			?gbrfsx	
general tridiagonal	?gttrf		?gttrs	?gtcon	?gtrfs	
Hermitian positive-definite	?potrf	?poequ,	?potrs	?pocon	?porfs,	?potri
		?poequb			?porfsx	
Hermitian positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
Hermitian positive-definite, RFP storage	?pftrf		?pftrs			?pftri
Hermitian positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
Hermitian positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
Hermitian indefinite	?hetrf	?heequb	?hetrs	?hecon	?herfs,	?hetri
	?hetrf_rook		?hetrs_rook	?hecon_rook	?herfsx	?hetri_rook
	?hetrf_rk		?hetrs2	?hecon_3		?hetri2
	?hetrf_aa		?hetrs_3			?hetri2x
			?hetrs_aa			?hetri_3
symmetric indefinite	?sytrf	?syequb	?sytrs	?sycon	?syrfs,	?sytri
	?sytrf_rook		?sytrs_rook	?sycon_rook	?syrfsx	?sytri_rook
	?sytrf_rk		?sytrs2	?sycon_3		?sytri2
			?sytrs3			?sytri2x
						?sytri_3
Hermitian indefinite, packed storage	?hptrf		?hptrs	?hpcon	?hprfs	?hptri
symmetric indefinite, packed storage	?sptrf		?sptrs	?spcon	?sprfs	?sptri
	mkl_?spffrt2, mkl_?spffrtx					
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprfs	?tptri
triangular, RFP storage						?tftri
triangular band			?tbtrs	?tbcon	?tbrfs	

In the table above, ? stands for c (single precision complex) or z (double precision complex) for FORTRAN 77 interface.

Matrix Factorization: LAPACK Computational Routines

This section describes the LAPACK routines for matrix factorization. The following factorizations are supported:

- *LU* factorization
- Cholesky factorization of real symmetric positive-definite matrices
- Cholesky factorization of real symmetric positive-definite matrices with pivoting
- Cholesky factorization of Hermitian positive-definite matrices
- Cholesky factorization of Hermitian positive-definite matrices with pivoting
- Bunch-Kaufman factorization of real and complex symmetric matrices
- Bunch-Kaufman factorization of Hermitian matrices.

You can compute:

- the *LU* factorization using full and band storage of matrices
- the Cholesky factorization using full, packed, RFP, and band storage
- the Bunch-Kaufman factorization using full and packed storage.

?getrf
Computes the LU factorization of a general m-by-n matrix.

Syntax

```
call sgetrf( m, n, a, lda, ipiv, info )
call dgetrf( m, n, a, lda, ipiv, info )
call cgetrf( m, n, a, lda, ipiv, info )
call zgetrf( m, n, a, lda, ipiv, info )
call getrf( a [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the *LU* factorization of a general *m*-by-*n* matrix *A* as

```
A = P*L*U,
```

where *P* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and *U* is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

m — INTEGER. The number of rows in the matrix *A* ($m \geq 0$).

n — INTEGER. The number of columns in *A*; $n \geq 0$.

a — REAL for sgetrf
DOUBLE PRECISION for dgetrf
COMPLEX for cgetrf
DOUBLE COMPLEX for zgetrf.
Array, size *lda* by * . Contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).

lda — INTEGER. The leading dimension of array *a*.

Output Parameters

a — Overwritten by *L* and *U*. The unit diagonal elements of *L* are not stored.

ipiv — INTEGER.

	Array, size at least `max(1,min(m, n))`. Contains the pivot indices; for `1 ≤i≤ min(m, n)`, row *i* was interchanged with row `ipiv(i)`.
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `getrf` interface are as follows:

a	Holds the matrix *A* of size (*m*, *n*).
ipiv	Holds the vector of length min(*m*, *n*).

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \le c(\min(m,n))\varepsilon P|L||U|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$(2/3)n^3$	If $m = n$,
$(1/3)n^2(3m-n)$	If $m>n$,
$(1/3)m^2(3n-m)$	If $m<n$.

The number of operations for complex flavors is four times greater.

After calling this routine with $m = n$, you can call the following:

`?getrs`	to solve $A*X = B$ or $A^TX = B$ or $A^HX = B$
`?gecon`	to estimate the condition number of *A*
`?getri`	to compute the inverse of *A*.

See Also

`mkl_progress`

Matrix Storage Schemes

?getrf_batch
Computes the LU factorization for 1 or more groups of general m-by-n matrices.

Syntax

```
call sgetrf_batch(m_array, n_array, A_array, lda_array, ipiv_array, group_size, group_count, info_array)
call dgetrf_batch(m_array, n_array, A_array, lda_array, ipiv_array, group_size, group_count, info_array)
call cgetrf_batch(m_array, n_array, A_array, lda_array, ipiv_array, group_size, group_count, info_array)
call zgetrf_batch(m_array, n_array, A_array, lda_array, ipiv_array, group_size, group_count, info_array)
```

Include Files

```
mkl.fi
```

Description

The `?getrf_batch` routines are similar to the `?getrf` counterparts, but instead compute the LU factorization for a group of general *m*-by-*n* matrices, processing one or more groups at once. Each group contains matrices with the same parameters.

The operation is defined as

```
i = 1
for g = 1 … group_count
    m_g n_g and lda_g in m_array(g), n_array(g) and lda_array(g)
    for j = 1 … group_size(g)
            A_i, ipiv_i in A_array(i), ipiv_array(i)
            A_i := P_i * L_i * U_i
            i = i + 1
    end for
end for
```

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m_g > n_g$) and U_i is upper triangular (upper trapezoidal if $m_g < n_g$). These routines use partial pivoting, with row interchanges.

A_i represents matrices stored at the addresses pointed to by *A_array*. The dimensions of each matrix is m_g-by-n_g, where m_g and n_g are the g-th elements of *m_array* and *n_array*, respectively. Similarly, $ipiv_i$ represents the pivot arrays stored at addresses pointed to by *ipiv_array*, where the size of the pivoting arrays is min(m_g, n_g).

The number of entries in *A_array* and *ipiv_array* is *total_batch_count*, which is equal to the sum of all the entries in the array *group_size*.

Refer to ?getrf for a detailed description of the LU factorization of general matrices.

Input Parameters

m_array

INTEGER. Array of size *group_count*. For the group g, m_g = *m_array*(*g*) specifies the number of rows of the matrices A_i in group g.

The value of each element of *m_array* must be at least zero.

n_array

INTEGER. Array of size *group_count*. For the group g, n_g = *n_array*(*g*) specifies the number of columns of the matrices A_i in group g.

The value of each element of *n_array* must be at least zero.

A_array
`INTEGER*8` for Intel® 64 architecture

`INTEGER*4` for IA-32 architecture

Array, size *total_batch_count*, of pointers to arrays used to store A_i matrices.

lda_array
`INTEGER`. Array of size `group_count`. For group *g*, lda_g = `lda_array(g)` specifies the leading dimension of the matricies A_i in group *g*, as declared in the calling (sub)program.

The value of lda_g must be at least max(1, m_g).

group_count
`INTEGER`.

Specifies the number of groups. Must be at least 0.

group_size
`INTEGER`.

Array of size `group_count`. The element `group_size(g)` specifies the number of matrices in group *g*. Each element in `group_size` must be at least 0.

Output Parameters

A_array
Output array, overwritten by the *total_batch_count* LU-factored matrices. Each matrix A_i is overwritten by L_i and U_i. The unit diagonal elements of L_i are not stored.

ipiv_array
`INTEGER*8` for Intel® 64 architecture

`INTEGER*4` for IA-32 architecture

Array, size *total_batch_count*, of pointers to the pivot arrays associated with the LU-factored A_i matrices.

info_array
`INTEGER`.

Array of size *total_batch_count*, which reports the factorization status for each matrix.

If *info(i)* = 0, the execution is successful for A_i.

If *info(i)* = -*j*, the *j*-th parameter had an illegal value for A_i.

If *info(i)* = *j*, the *j*-th diagonal element of U_i is 0. The factorization has been completed, but U_i is exactly singular. Division by 0 will occur if you use the factor U_i for solving a system of linear equations.

Related Information

Refer to `?getrf_batch_strided` , which computes the LU factorization for a group of general *m*-by-*n* matrices that are allocated at a constant stride from each other in the same contiguous block of memory.

?getrf_batch_strided
Computes the LU factorization of a group of general m-by-n matrices that are stored at a constant stride from each other in a contiguous block of memory.

Syntax

```
call sgetrf_batch_strided(m, n, A, lda, stride_a, ipiv, stride_ipiv, batch_size, info)
```

```
call dgetrf_batch_strided(m, n, A, lda, stride_a, ipiv, stride_ipiv, batch_size, info)
call cgetrf_batch_strided(m, n, A, lda, stride_a, ipiv, stride_ipiv, batch_size, info)
call zgetrf_batch_strided(m, n, A, lda, stride_a, ipiv, stride_ipiv, batch_size, info)
```

Include Files

mkl.fi

Description

The ?getrf_batch_strided routines are similar to the ?getrf counterparts, but instead compute the LU factorization for a group of general *m*-by-*n* matrices.

All matrices have the same parameters (matrix size, leading dimension) and are stored at constant *stride_a* from each other in a contiguous block of memory. Their respective pivot arrays associated with each of the LU-factored A_i matrices are stored at constant *stride_ipiv* from each other. The operation is defined as

```
for i = 0 … batch_size-1
    A_i is a matrix at offset i * stride_a from A
    ipiv_i is an array at offset i * stride_ipiv from ipiv
    A_i :=  P_i * L_i* U_i
end for
```

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U_i is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

Input Parameters

m

INTEGER. The number of rows in the *A* matrices: $m \geq 0$.

n

INTEGER. The number of columns in the *A* matrices: $n \geq 0$.

A

REAL for sgetrf_batch_strided

DOUBLE PRECISION for dgetrf_batch_strided

COMPLEX for cgetrf_batch_strided

DOUBLE COMPLEX for zgetrf_batch_strided

The *A* array of size at least *stride_a* * *batch_size* holding the A_i matrices.

lda

INTEGER. Specifies the leading dimension of the A_i matrices; *lda* ≥ max(1,*m*).

stride_a

INTEGER.

Stride between two consecutive A_i matrices; *stride_a* ≥ *lda* * *n*.

stride_ipiv

INTEGER.

Stride between two consecutive pivot arrays; *stride_ipiv* ≥ min(*m*, *n*).

batch_size

INTEGER.

Number of A_i matrices to be factorized. Must be at least 0.

Output Parameters

A	Array holding the LU-factored A_i matrices. Each matrix is overwritten by their respective L_i and U_i factors. The unit diagonal elements of *L* are not stored.
ipiv	INTEGER Array of size at least *stride_ipiv* * *batch_size* holding the pivot array associated with each of the LU-factored A_i matrices. The pivot array $ipiv_i$ contains the pivot indices associated with matrix A_i; for $1 \le j \le \min(m,n)$, row j was interchanged with row $ipiv_i(j)$.
info	INTEGER. Array of size at least *batch_size*, which reports the factorization status for each matrix. If *info(i)* = 0, the execution is successful for A_i. If *info(i)* = -*j*, the *j*-th parameter had an illegal value for A_i. If *info(i)* = *j*, the *j*-th diagonal element of U_i is 0. The factorization has been completed, but U_i is exactly singular. Division by 0 will occur if you use the factor U_i for solving a system of linear equations.

After calling this routine with m=n, you can call the following:

?getrs_batch_strided

to solve systems of linear equations of the form $A_i * X_i = B_i$ *or* $A_i^T * X_i = B_i$ *or* $A_i^H * X_i = B_i$ with the group of LU-factored matrices.

Related Information

See ?getrf_batch, which computes the LU factorization for a group of general m-by-n matrices that are allocated at a constant stride from each other in the same contiguous block of memory.

mkl_?getrfnp

Computes the LU factorization of a general m-by-n matrix without pivoting.

Syntax

```
call mkl_sgetrfnp( m, n, a, lda, info )
call mkl_dgetrfnp( m, n, a, lda, info )
call mkl_cgetrfnp( m, n, a, lda, info )
call mkl_zgetrfnp( m, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine computes the *LU* factorization of a general *m*-by-*n* matrix *A* as

```
A = L*U,
```

where *L* is lower triangular with unit-diagonal elements (lower trapezoidal if $m > n$) and *U* is upper triangular (upper trapezoidal if $m < n$). The routine does not use pivoting.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns in *A*; $n \geq 0$.
a	REAL for mkl_sgetrfnp DOUBLE PRECISION for mkl_dgetrfnp COMPLEX for mkl_cgetrfnp DOUBLE COMPLEX for mkl_zgetrfnp. Array, size *lda* by *. Contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The leading dimension of array *a*.

Output Parameters

a	Overwritten by *L* and *U*. The unit diagonal elements of *L* are not stored.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The approximate number of floating-point operations for real flavors is

$(2/3)n^3$	If $m = n$,
$(1/3)n^2(3m-n)$	If $m>n$,
$(1/3)m^2(3n-m)$	If $m<n$.

The number of operations for complex flavors is four times greater.

After calling this routine with $m = n$, you can call the following:

mkl_?getrinp	to compute the inverse of *A*

See Also

mkl_progress

Matrix Storage Schemes

?getrfnp_batch_strided

Computes the LU factorization, without pivoting, of a group of general m-by-n matrices that are stored at a constant stride from each other in a contiguous block of memory.

Syntax

```
call sgetrfnp_batch_strided(m, n, A, lda, stride_a, batch_size, info)
call dgetrfnp_batch_strided(m, n, A, lda, stride_a, batch_size, info)
call cgetrfnp_batch_strided(m, n, A, lda, stride_a, batch_size, info)
call zgetrfnp_batch_strided(m, n, A, lda, stride_a, batch_size, info)
```

Include Files

```
mkl.fi
```

Description

The `?getrfnp_batch_strided` routines are similar to the `?getrfnp` counterparts, but instead compute the LU factorization for a group of general *m*-by-*n* matrices.

All matrices `A` have the same parameters (matrix size, leading dimension) and are stored at constant `stride_a` from each other in a single block of memory. The operation is defined as

```
for i = 0 … batch_size-1
    A_i is a matrix at offset i * stride_a from A
    A_i :=  L_i* U_i
end for
```

where L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U_i is upper triangular (upper trapezoidal if $m < n$). The routine does not perform any pivoting.

Input Parameters

`m`	`INTEGER`. The number of rows in the `A` matrices: $m \geq 0$.
`n`	`INTEGER`. The number of columns in the A_i matrices: $n \geq 0$.

`A`	`REAL` for `sgetrfnp_batch_strided`
	`DOUBLE PRECISION` for `dgetrfnp_batch_strided`
	`COMPLEX` for `cgetrfnp_batch_strided`
	`DOUBLE COMPLEX` for `zgetrfnp_batch_strided`
	The `A` array of size at least `stride_a` * `batch_size` holding the A_i matrices.

`lda`	`INTEGER`. Specifies the leading dimension of the A_i matrices; `lda` ≥ max(1,`m`).
`stride_a`	`INTEGER`. Stride between two consecutive A_i matrices; `stride_a` ≥ `lda` * `n`.
`batch_size`	`INTEGER`. Number of A_i matrices to be factorized. Must be at least 0.

Output Parameters

A	Array holding the LU-factored A_i matrices. Each matrix is overwritten by their respective L_i and U_i factors. The unit diagonal elements of *L* are not stored.
info	INTEGER. Array of size at least *batch_size*, which reports the factorization status for each matrix. If *info(i)* = 0, the execution is successful for A_i. If *info(i)* = -*j*, the *j*-th parameter had an illegal value for A_i. If *info(i)* = *j*, the *j*-th diagonal element of U_i is 0. The factorization has been completed, but U_i is exactly singular. Division by 0 will occur if you use the factor U_i for solving a system of linear equations.

After calling this routine with *m*=*n*, you can call the following:

?getrsnp_batch_strided

to solve systems of linear equations of the form $A_i * X_i = B_i$ *or* $A_i^T * X_i = B_i$ *or* $A_i^H * X_i = B_i$ with the group of LU-factored matrices.

mkl_?getrfnpi
Performs LU factorization (complete or incomplete) of a general matrix without pivoting.

Syntax

```
call mkl_sgetrfnpi (m, n, nfact, a, lda, info)
call mkl_dgetrfnpi (m, n, nfact, a, lda, info )
call mkl_cgetrfnpi (m, n, nfact, a, lda, info )
call mkl_zgetrfnpi (m, n, nfact, a, lda, info )
call mkl_getrfnpi ( a [, nfact] [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the LU factorization of a general *m*-by-*n* matrix *A* without using pivoting. It supports incomplete factorization. The factorization has the form:

*A = L*U*,

where *L* is lower triangular with unit diagonal elements (lower trapezoidal if *m* > *n*) and *U* is upper triangular (upper trapezoidal if *m* < *n*).

Incomplete factorization has the form:

$$A = L * U + \tilde{A}$$

where *L* is lower trapezoidal with unit diagonal elements, *U* is upper trapezoidal, and

$$\tilde{A}$$

is the unfactored part of matrix *A*. See the application notes section for further details.

NOTE
Use `?getrf` if it is possible that the matrix is not diagonal dominant.

Input Parameters

The data types are given for the Fortran interface.

m — INTEGER. The number of rows in matrix *A*; *m*≥ 0.

n — INTEGER. The number of columns in matrix *A*; *n*≥ 0.

nfact — INTEGER. The number of rows and columns to factor; 0 ≤*nfact*≤ min(*m*, *n*). Note that if *nfact* < min(*m*, *n*), incomplete factorization is performed.

a — `REAL` for `mkl_sgetrfnpi`

`DOUBLE PRECISION` for `mkl_dgetrfnpi`

`COMPLEX` for `mkl_cgetrfnpi`

`DOUBLE COMPLEX` for `mkl_zgetrfnpi`

Array of size `(lda,*)`. Contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).

lda — INTEGER. The leading dimension of array *a*. *lda*≥ max(1, *m*).

Output Parameters

a — Overwritten by *L* and *U*. The unit diagonal elements of *L* are not stored.

When incomplete factorization is specified by setting *nfact* < min(*m*, *n*), *a* also contains the unfactored submatrix

$$\tilde{A}_{22}$$

. See the application notes section for further details.

info — INTEGER. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, u_{ii} is 0. The requested factorization has been completed, but *U* is exactly singular. Division by 0 will occur if factorization is completed and factor *U* is used for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran 95 Interface Conventions.

Specific details for the routine `getrf` interface are as follows:

a — Holds the matrix *A* of size (*m*, *n*).

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix *A* + *E*, with

$$|E| \leq c(\min(m, n))\varepsilon|L||U|$$

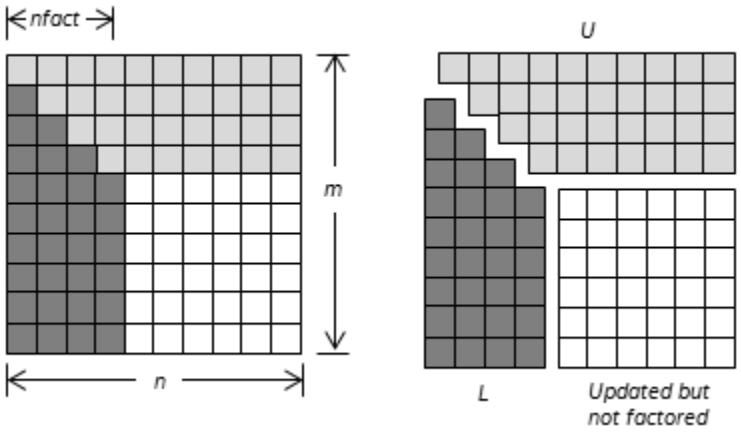
where `c(n)` is a modest linear function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$(2/3)n^3$	If `m = n` = `nfact`
$(1/3)n^2(3m-n)$	If `m>n = nfact`
$(1/3)m^2(3n-m)$	If `m = nfact<n`
$(2/3)n^3 - (n-nfact)^3$	If `m = n`, `nfact< min(m, n)`
$(1/3)(n^2(3m-n) - (n-nfact)^2(3m - 2nfact - n))$	If `m>n > nfact`
$(1/3)(m^2(3n-m) - (m-nfact)^2(3n - 2nfact - m))$	If `nfact < m < n`

The number of operations for complex flavors is four times greater.

When incomplete factorization is specified, the first `nfact` rows and columns are factored, with the update of the remaining rows and columns of *A* as follows:



If matrix *A* is represented as a block 2-by-2 matrix:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where

- A_{11} is a square matrix of order `nfact`,
- A_{21} is an (`m` - `nfact`)-by-`nfact` matrix,
- A_{12} is an `nfact`-by-(`n` - `nfact`) matrix, and
- A_{22} is an (`m` - `nfact`)-by-(`n` - `nfact`) matrix.

The result is

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} \cdot \begin{bmatrix} U_1 & U_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & \tilde{A}_{22} \end{bmatrix}$$

L_1 is a lower triangular square matrix of order `nfact` with unit diagonal and U_1 is an upper triangular square matrix of order `nfact`. L_1 and U_1 result from LU factorization of matrix A_{11}: $A_{11} = L_1U_1$.

L_2 is an (m - $nfact$)-by-$nfact$ matrix and $L_2 = A_{21}U_1^{-1}$. U_2 is an $nfact$-by-(n - $nfact$) matrix and $U_2 = L_1^{-1}A_{12}$.

$$\tilde{A}_{22}$$

is an (m - $nfact$)-by-(n - $nfact$) matrix and

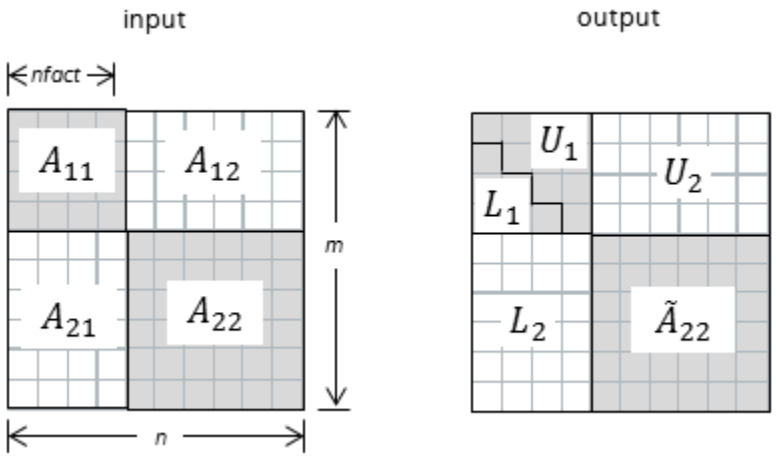
$$\tilde{A}_{22}$$

$= A_{22} - L_2U_2$.

On exit, elements of the upper triangle U_1 are stored in place of the upper triangle of block A_{11} in array a; elements of the lower triangle L_1 are stored in the lower triangle of block A_{11} in array a (unit diagonal elements are not stored). Elements of L_2 replace elements of A_{21}; U_2 replaces elements of A_{12} and

$$\tilde{A}_{22}$$

replaces elements of A_{22}.



?getrf2
Computes LU factorization using partial pivoting with row interchanges.

Syntax

```
call sgetrf2 (m, n, a, lda, ipiv, info )
call dgetrf2 (m, n, a, lda, ipiv, info )
call cgetrf2 (m, n, a, lda, ipiv, info )
call zgetrf2 (m, n, a, lda, ipiv, info )
```

Include Files

- `mkl.fi`

Description

`?getrf2` computes an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges.

The factorization has the form

$A = P * L * U$

where *P* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and *U* is upper triangular (upper trapezoidal if $m < n$).

This is the recursive version of the algorithm. It divides the matrix into four submatrices:

$$A = \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix}$$

where *A11* is *n1* by *n1* and *A22* is *n2* by *n2* with *n1* = min(m, n), and *n2* = n - *n1*.

The subroutine calls itself to factor $\begin{pmatrix} A11 \\ A12 \end{pmatrix}$,

do the swaps on $\begin{pmatrix} A12 \\ A22 \end{pmatrix}$, solve *A12*, update *A22*, then it calls itself to factor *A22* and do the swaps on *A21*.

Input Parameters

m	INTEGER. The number of rows of the matrix A. $m >= 0$.
n	INTEGER. The number of columns of the matrix A. $n >= 0$.

a	`REAL` for `sgetrf2`
	`DOUBLE PRECISION` for `dgetrf2`
	`COMPLEX` for `cgetrf2`
	`DOUBLE COMPLEX` for `zgetrf2`
	Array, size (*lda*,*n*).
	On entry, the *m*-by-*n* matrix to be factored.

lda	INTEGER. The leading dimension of the array *a*. *lda* >= max(1,*m*).

Output Parameters

a	On exit, the factors *L* and *U* from the factorization *A* = *P* * *L* * *U*; the unit diagonal elements of *L* are not stored.
ipiv	INTEGER. Array, size (min(*m*,*n*)). The pivot indices; for 1 <= *i* <= min(*m*,*n*), row *i* of the matrix was interchanged with row *ipiv(i)*.
info	INTEGER. = 0: successful exit. < 0: if info = -*i*, the *i*-th argument had an illegal value. > 0: if info = *i*, $U_{i,i}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?getri_oop_batch
Computes the inverses for 1 or more groups of LU factored, n-by-n matrices.

Syntax

```
call sgetri_oop_batch(n_array, A_array, lda_array, ipiv_array, Ainv_array, ldainv_array, group_count, group_size, info_array)
```

```
call dgetri_oop_batch(n_array, A_array, lda_array, ipiv_array, Ainv_array, ldainv_array,
group_count, group_size, info_array)
```

```
call cgetri_oop_batch(n_array, A_array, lda_array, ipiv_array, Ainv_array, ldainv_array,
group_count, group_size, info_array)
```

```
call zgetri_oop_batch(n_array, A_array, lda_array, ipiv_array, Ainv_array, ldainv_array,
group_count, group_size, info_array)
```

Include Files

mkl.fi

Description

The ?getri_oop_batch routines are similar to the ?getri counterparts, but instead compute the inverses for groups of n-by-n LU factored matrices, processing one or more groups at once. Each group contains matrices with the same parameters.

The operation is defined as

```
i = 1
for g = 1 … group_count
    n_g and lda_g in n_array(g) and lda_array(g)
    for j = 1 … group_size(g)
    A_i, Ainv_i, ipiv_i in A_array(i), Ainv_array(i), ipiv_array(i)
    Ainv_i :=  inv(P_i * L_i* U_i)
    i = i + 1
    end for
end for
```

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements and U_i is upper triangular. These routines use partial pivoting, with row interchanges.

A_i and $Ainv_i$ represents matrices stored at the addresses pointed to by *A_array* and *Ainv_array*. The dimensions of each matrix is n_g-by-n_g, where n_g is the g-th elements of *n_array*. Similarly, $ipiv_i$ represents the pivot arrays stored at addresses pointed to by *ipiv_array*, where the size of the pivoting arrays is n_g.

The number of entries in *A_array*, *Ainv_array* and *ipiv_array* is *total_batch_count*, which is equal to the sum of all the entries in the array *group_size*.

Refer to ?getri for a detailed description of the inversion of LU factorized matrices.

Input Parameters

n_array

INTEGER. Array of size *group_count*. For the group g, n_g = *n_array*(*g*) specifies the order of the matrices A_i in group g.

The value of each element of *n_array* must be at least zero.

A_array

INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA-32 architecture

Array, size *total_batch_count*, of pointers to the A_i matrices.

lda_array

INTEGER. Array of size *group_count*. For group g, lda_g = *lda_array*(*g*) specifies the leading dimension of the matricies A_i in group g, as declared in the calling (sub)program.

The value of lda_g must be at least max(1, n_g).

ipiv_array

INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA-32 architecture

Array, size *total_batch_count*, of pointers to the pivot arrays associated with the LU-factored A_i matrices, as returned by ?getrf_batch.

group_count INTEGER.

Specifies the number of groups. Must be at least 0.

group_size INTEGER.

Array of size *group_count*. The element *group_size*(*g*) specifies the number of matrices in group *g*. Each element in *group_size* must be at least 0.

Output Parameters

Ainv_array INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA-32 architecture

Array, size *total_batch_count*, of pointers to the $Ainv_i$ matrices.

Each matrix is overwritten by the n_g-by-n_g matrix $inv(A_i)$.

ldainv_array INTEGER.

Array of size *group_count*. For group *g*, $ldainv_g$ = *ldainv_array*(*g*) specifies the leading dimension of the matrices $Ainv_i$ in group *g*.

The value of $ldainv_g$ must be at least max(1, n_g).

info_array INTEGER.

Array of size *total_batch_count*, which reports the inversion status for each matrix.

If *info(i)* = 0, the execution is successful for A_i.

If *info(i)* = -*j*, the *j*-th parameter had an illegal value for A_i.

If *info(i)* = *j*, the *j*-th diagonal element of the factor U_i is 0, U_i is singular, and the inversion could not be completed.

Related Information

Refer to ?getri_oop_batch_strided , which computes inverses for a group of *n*-by-*n* matrices that are allocated at a constant stride from each other in the same contiguous block of memory.

?getri_oop_batch_strided
Computes the inverses of a group of LU factored matrices that are stored at a constant stride from each other in a contiguous block of memory.

Syntax

```
call sgetri_oop_batch_strided(n, A, lda, stride_a, ipiv, stride_ipiv, Ainv, ldainv, stride_ainv, batch_size, info)
```

```
call dgetri_oop_batch_strided(n, A, lda, stride_a, ipiv, stride_ipiv, Ainv, ldainv, stride_ainv, batch_size, info)
```

```
call cgetri_oop_batch_strided(n, A, lda, stride_a, ipiv, stride_ipiv, Ainv, ldainv, stride_ainv, batch_size, info)
```

```
call zgetri_oop_batch_strided(n, A, lda, stride_a, ipiv, stride_ipiv, Ainv, ldainv,
stride_ainv, batch_size, info)
```

Include Files

```
mkl.fi
```

Description

The `?getri_oop_batch_strided` routines are similar to the `?getri` counterparts, but instead compute the inverses for a group of LU factored matrices.

All matrices have the same parameters (matrix size, leading dimension) and are stored at constant *stride_a* from each other in a contiguous block of memory. The output arrays are stored at constant *stride_ainv* from each other in a contiguous block of memory. Their respective pivot arrays associated with each of the LU-factored A_i matrices are stored at constant *stride_ipiv* from each other.

The operation is defined as

```
for i = 0 … batch_size-1
    A_i is a matrix at offset i * stride_a from A
    ipiv_i is an array at offset i * stride_ipiv from ipiv
    Ainv_i is a matrix at offset i * stride_ainv from Ainv
    Ainv_i :=  inv(P_i * L_i* U_i)
end for
```

where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m>n$), and U_i is upper triangular (upper trapezoidal if $m<n$). The routine uses partial pivoting, with row interchanges.

Input Parameters

n
: `INTEGER`. The number of columns in the A_i matrices; $n \geq 0$.

A
: REAL for `sgetri_oop_batch_strided`
: DOUBLE PRECISION for `dgetri_oop_batch_strided`
: COMPLEX for `cgetri_oop_batch_strided`
: DOUBLE COMPLEX for `zgetri_oop_batch_strided`
: The *A* array of size at least *stride_a* * *batch_size* holding the A_i matrices.

lda
: `INTEGER`. Specifies the leading dimension of the A_i matrices; *lda* ≥ *n*.

stride_a
: `INTEGER`. Stride between two consecutive A_i matrices; ; *stride_a* ≥ *lda* * *n*.

ipiv
: `INTEGER`.
: Array of size at least *stride_ipiv* * *batch_size*, holding the pivot array associated with each of the LU-factored A_i matrices, as returned by `?getrf_batch_strided`.

stride_ipiv
: `INTEGER`.
: Stride between two consecutive pivot arrays; *stride_ipiv* ≥ *n*.

ldainv
: `INTEGER`. Specifies the leading dimension of the $Ainv_i$ matrices; *ldainv* ≥ *n*.

stride_ainv
: `INTEGER`. Stride between two consecutive $Ainv_i$ matrices; ; *stride_ainv* ≥ *ldainv* * *n*.

batch_size
: `INTEGER`.

Number of A_i matrices to be factorized. Must be at least 0.

Output Parameters

Ainv — Array holding the inverses $Ainv_i$ matrices. Each matrix is overwritten by their respective L_i and $U_i$factors. The unit diagonal elements of L are not stored.

info — INTEGER.

Array of size *batch_size*, which reports the factorization status for each matrix.

If *info(i)* = 0, the execution is successful for A_i.

If *info(i)* = -*j*, the *j*-th parameter had an illegal value for A_i.

If *info(i)* = *j*, the *j*-th diagonal element of the factor U_i is 0, U_i is exactly singular. Division by - will occur if you use the factor U_i to solve a system of linear equations.

?gbtrf

Computes the LU factorization of a general m-by-n band matrix.

Syntax

```
call sgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call gbtrf( ab [,kl] [,m] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the *LU* factorization of a general *m*-by-*n* band matrix *A* with *kl* non-zero subdiagonals and *ku* non-zero superdiagonals, that is,

```
A = P*L*U,
```

where *P* is a permutation matrix; *L* is lower triangular with unit diagonal elements and at most *kl* non-zero elements in each column; *U* is an upper triangular band matrix with *kl* + *ku* superdiagonals. The routine uses partial pivoting, with row interchanges (which creates the additional *kl* superdiagonals in *U*).

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

m — INTEGER. The number of rows in matrix *A*; $m \geq 0$.

n — INTEGER. The number of columns in matrix *A*; $n \geq 0$.

kl — INTEGER. The number of subdiagonals within the band of *A*; $kl \geq 0$.

ku — INTEGER. The number of superdiagonals within the band of *A*; $ku \geq 0$.

ab — REAL for sgbtrf

DOUBLE PRECISION for dgbtrf

COMPLEX for cgbtrf

DOUBLE COMPLEX for zgbtrf.

Array, size *ldab* by *.

The array *ab* contains the matrix *A* in band storage, in rows *kl* + 1 to 2**kl* + *ku* + 1; rows 1 to *kl* of the array need not be set. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

ab(*kl* + *ku* + 1 + *i* – *j*, *j*) = *a*(*i*,*j*) for max(1,*j*–*ku*) ≤*i*≤ min(*m*,*j*+*kl*).

ldab — INTEGER. The leading dimension of the array *ab*. (*ldab*≥ 2**kl* + *ku* + 1)

Output Parameters

ab — Overwritten by *L* and *U*. *U* is stored as an upper triangular band matrix with *kl* + *ku* superdiagonals in rows 1 to *kl* + *ku* + 1, and the multipliers used during the factorization are stored in rows *kl* + *ku* + 2 to 2**kl* + *ku* + 1.

See Application Notes below for further details.

ipiv — INTEGER.

Array, size at least max(1,min(*m*, *n*)). The pivot indices; for 1 ≤*i*≤ min(*m*, *n*) , row *i* was interchanged with row *ipiv*(*i*).

info — If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gbtrf interface are as follows:

ab — Holds the array *A* of size (2**kl*+*ku*+1,*n*).

ipiv — Holds the vector of length min(*m*,*n*).

kl — If omitted, assumed *kl* = *ku*.

ku	Restored as *ku* = *lda*-2**kl*-1.
m	If omitted, assumed *m* = *n*.

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \le c(kl+ku+1)\ \varepsilon P|L||U|$$

$c(k)$ is a modest linear function of *k*, and ε is the machine precision.

The total number of floating-point operations for real flavors varies between approximately $2n(ku+1)kl$ and $2n(kl+ku+1)kl$. The number of operations for complex flavors is four times greater. All these estimates assume that *kl* and *ku* are much less than $\min(m,n)$.

The band storage scheme is illustrated by the following example, when *m* = *n* = 6, *kl* = 2, *ku* = 1:

on entry

$$\begin{bmatrix} * & * & * & + & + & + \\ * & * & + & + & + & + \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{bmatrix}$$

on exit

$$\begin{bmatrix} * & * & * & u_{14} & u_{25} & u_{36} \\ * & * & u_{13} & u_{24} & u_{35} & u_{46} \\ * & u_{12} & u_{23} & u_{34} & u_{45} & u_{56} \\ u_{11} & u_{22} & u_{33} & u_{44} & u_{55} & u_{66} \\ m_{21} & m_{32} & m_{43} & m_{54} & m_{65} & * \\ m_{31} & m_{42} & m_{53} & m_{64} & * & * \end{bmatrix}$$

Elements marked * are not used; elements marked + need not be set on entry, but are required by the routine to store elements of *U* because of fill-in resulting from the row interchanges.

After calling this routine with *m* = *n*, you can call the following routines:

gbtrs	to solve *A*X* = *B* or A^T*X* = *B* or A^H*X* = *B*
gbcon	to estimate the condition number of *A*.

See Also

mkl_progress

Matrix Storage Schemes

?gttrf
Computes the LU factorization of a tridiagonal matrix.

Syntax

```
call sgttrf( n, dl, d, du, du2, ipiv, info )
call dgttrf( n, dl, d, du, du2, ipiv, info )
call cgttrf( n, dl, d, du, du2, ipiv, info )
call zgttrf( n, dl, d, du, du2, ipiv, info )
call gttrf( dl, d, du, du2 [, ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the *LU* factorization of a real or complex tridiagonal matrix *A* using elimination with partial pivoting and row interchanges.

The factorization has the form

```
A = L*U,
```

where *L* is a product of permutation and unit lower bidiagonal matrices and *U* is upper triangular with nonzeroes in only the main diagonal and first two superdiagonals.

Input Parameters

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

dl, *d*, *du*

REAL for sgttrf

DOUBLE PRECISION for dgttrf

COMPLEX for cgttrf

DOUBLE COMPLEX for zgttrf.

Arrays containing elements of *A*.

The array *dl* of dimension (*n* – 1) contains the subdiagonal elements of *A*.

The array *d* of dimension *n* contains the diagonal elements of *A*.

The array *du* of dimension (*n* – 1) contains the superdiagonal elements of *A*.

Output Parameters

dl

Overwritten by the (*n*–1) multipliers that define the matrix *L* from the *LU* factorization of *A*. The matrix *L* has unit diagonal elements, and the (*n*–1) elements of *dl* form the subdiagonal. All other elements of *L* are zero.

d

Overwritten by the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

du

Overwritten by the (*n*–1) elements of the first superdiagonal of *U*.

du2

REAL for sgttrf

DOUBLE PRECISION for dgttrf

COMPLEX for cgttrf

DOUBLE COMPLEX for zgttrf.

Array, dimension (*n* –2). On exit, *du2* contains (*n*–2) elements of the second superdiagonal of *U*.

ipiv

INTEGER.

	Array, dimension (*n*). The pivot indices: for $1 \le i \le n$, row *i* was interchanged with row *ipiv*(*i*). *ipiv*(*i*) is always *i* or *i*+1; *ipiv*(*i*) = *i* indicates a row interchange was not required.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gttrf interface are as follows:

dl	Holds the vector of length (*n*-1).
d	Holds the vector of length *n*.
du	Holds the vector of length (*n*-1).
du2	Holds the vector of length (*n*-2).
ipiv	Holds the vector of length *n*.

Application Notes

?gbtrs	to solve *A*X = B* or $A^T*X = B$ or $A^H*X = B$
?gbcon	to estimate the condition number of *A*.

?dttrfb

Computes the factorization of a diagonally dominant tridiagonal matrix.

Syntax

```
call sdttrfb( n, dl, d, du, info )
call ddttrfb( n, dl, d, du, info )
call cdttrfb( n, dl, d, du, info )
call zdttrfb( n, dl, d, du, info )
call dttrfb( dl, d, du [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The ?dttrfb routine computes the factorization of a real or complex tridiagonal matrix *A* with the BABE (Burning At Both Ends) algorithm without pivoting. The factorization has the form

$A = L_1 * U * L_2$

where

- L_1 and L_2 are unit lower bidiagonal with *k* and *n* - *k* - 1 subdiagonal elements, respectively, where *k* = *n*/2, and
- *U* is an upper bidiagonal matrix with nonzeroes in only the main diagonal and first superdiagonal.

Input Parameters

n	INTEGER. The order of the matrix *A*; $n \geq 0$.
dl, *d*, *du*	REAL for sdttrfb DOUBLE PRECISION for ddttrfb COMPLEX for cdttrfb DOUBLE COMPLEX for zdttrfb. Arrays containing elements of *A*. The array *dl* of dimension (*n* – 1) contains the subdiagonal elements of *A*. The array *d* of dimension *n* contains the diagonal elements of *A*. The array *du* of dimension (*n* – 1) contains the superdiagonal elements of *A*.

Output Parameters

dl	Overwritten by the (*n* –1) multipliers that define the matrix *L* from the *LU* factorization of *A*.
d	Overwritten by the *n* diagonal element reciprocals of the upper triangular matrix *U* from the factorization of *A*.
du	Overwritten by the (*n*–1) elements of the superdiagonal of *U*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

A diagonally dominant tridiagonal system is defined such that $|d_i| > |dl_{i-1}| + |du_i|$ for any *i*:

$1 < i < n$, and $|d_1| > |du_1|$, $|d_n| > |dl_{n-1}|$

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems are free from the numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see ?gttrf). The diagonally dominant systems are much faster than the canonical systems.

NOTE

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
- Applying the `?dttrfb` factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.

?potrf
Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

Syntax

```
call spotrf( uplo, n, a, lda, info )
call dpotrf( uplo, n, a, lda, info )
call cpotrf( uplo, n, a, lda, info )
call zpotrf( uplo, n, a, lda, info )
call potrf( a [, uplo] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix *A*:

$A = U^T * U$ for real data, $A = U^H * U$ for complex data	if *uplo*='U'
$A = L * L^T$ for real data, $A = L * L^H$ for complex data	if *uplo*='L'

where *L* is a lower triangular matrix and *U* is upper triangular.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:

If *uplo* = `'U'`, the array *a* stores the upper triangular part of the matrix *A*, and the strictly lower triangular part of the matrix is not referenced.

If *uplo* = `'L'`, the array *a* stores the lower triangular part of the matrix *A*, and the strictly upper triangular part of the matrix is not referenced.

n	INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.
a	REAL for spotrf DOUBLE PRECISION for dpotrf COMPLEX for cpotrf DOUBLE COMPLEX for zpotrf. Array, size (*lda*,*). The array *a* contains either the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The leading dimension of *a*.

Output Parameters

a	The upper or lower triangular part of *a* is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine potrf interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', the computed factor *U* is the exact factor of a perturbed matrix *A* + *E*, where

$$|E| \le c(n)\varepsilon |U^H| |U|, |e_{ij}| \le c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?potrs	to solve *A***X* = *B*

?pocon	to estimate the condition number of *A*
?potri	to compute the inverse of *A*.

See Also

mkl_progress

Matrix Storage Schemes

?potrf2
Computes Cholesky factorization using a recursive algorithm.

Syntax

```
call spotrf2(uplo, n, a, lda, info)
call dpotrf2(uplo, n, a, lda, info)
call cpotrf2(uplo, n, a, lda, info)
call zpotrf2(uplo, n, a, lda, info)
```

Include Files

- mkl.fi

Description

?potrf2 computes the Cholesky factorization of a real or complex symmetric positive definite matrix *A* using the recursive algorithm.

The factorization has the form

for real flavors:

A = U^T * *U*, if *uplo* = 'U', or

A = *L* * L^T, if *uplo* = 'L',

for complex flavors:

A = U^H * *U*, if *uplo* = 'U',

or *A* = *L* * L^H, if *uplo* = 'L',

where *U* is an upper triangular matrix and *L* is lower triangular.

This is the recursive version of the algorithm. It divides the matrix into four submatrices:

$$A = \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix}$$

where *A11* is *n1* by *n1* and *A22* is *n2* by *n2*, with *n1* = *n*/2 and *n2* = *n*-*n1*.

The subroutine calls itself to factor *A11*. Update and scale *A21* or *A12*, update *A22* then call itself to factor *A22*.

Input Parameters

uplo	CHARACTER*1. = 'U': Upper triangle of *A* is stored; = 'L': Lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*.

n ≥ 0.

a

REAL for spotrf2

DOUBLE PRECISION for dpotrf2

COMPLEX for cpotrf2

DOUBLE COMPLEX for zpotrf2

Array, size (*lda*, *n*).

On entry, the symmetric matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda

INTEGER. The leading dimension of the array *a*.

lda ≥ max(1,*n*).

Output Parameters

a

On exit, if *info* = 0, the factor *U* or *L* from the Cholesky factorization.

For real flavors:

$A = U^T*U$ or $A = L*L^T$;

For complex flavors:

$A = U^H*U$ or $A = L*L^H$.

info

INTEGER. = 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument had an illegal value

> 0: if *info* = *i*, the leading minor of order *i* is not positive definite, and the factorization could not be completed.

?pstrf

Computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix.

Syntax

```
call spstrf( uplo, n, a, lda, piv, rank, tol, work, info )
call dpstrf( uplo, n, a, lda, piv, rank, tol, work, info )
call cpstrf( uplo, n, a, lda, piv, rank, tol, work, info )
call zpstrf( uplo, n, a, lda, piv, rank, tol, work, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix. The form of the factorization is:

$P^T * A * P = U^T * U$, if *uplo* = 'U' for real flavors,
$P^T * A * P = U^H * U$, if *uplo* = 'U' for complex flavors,
$P^T * A * P = L * L^T$, if *uplo* = 'L' for real flavors,
$P^T * A * P = L * L^H$, if *uplo* = 'L' for complex flavors,

where *P* is a permutation matrix stored as vector *piv*, and *U* and *L* are upper and lower triangular matrices, respectively.

This algorithm does not attempt to check that *A* is positive semidefinite. This version of the algorithm calls level 3 BLAS.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*, and the strictly lower triangular part of the matrix is not referenced.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*, and the strictly upper triangular part of the matrix is not referenced.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

a

REAL for spstrf

DOUBLE PRECISION for dpstrf

COMPLEX for cpstrf

DOUBLE COMPLEX for zpstrf.

Array *a*, size (*lda*,*). The array *a* contains either the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*).

work

REAL for spstrf and cpstrf

DOUBLE PRECISION for dpstrf and zpstrf.

work(*) is a workspace array. The dimension of *work* is at least max(1,2**n*).

tol

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

User defined tolerance. If *tol* < 0, then $n*\varepsilon*\max(A_{k,k})$, where ε is the machine precision, will be used (see Error Analysis for the definition of machine precision). The algorithm terminates at the (*k*-1)-st step, if the pivot $\leq tol$.

lda

INTEGER. The leading dimension of *a*; at least max(1, *n*).

Output Parameters

a	If *info* = 0, the factor U or L from the Cholesky factorization is as described in *Description*.
piv	INTEGER. Array, size at least max(1, *n*). The array *piv* is such that the nonzero entries are $P_{piv(k),k}$ ($1 \le k \le n$).
rank	INTEGER. The rank of *a* given by the number of steps the algorithm completed.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*k*, the *k*-th argument had an illegal value. If *info* > 0, the matrix *A* is either rank deficient with a computed rank as returned in *rank*, or is not positive semidefinite.

See Also

Matrix Storage Schemes

?pftrf
Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using the Rectangular Full Packed (RFP) format .

Syntax

```
call spftrf( transr, uplo, n, a, info )
call dpftrf( transr, uplo, n, a, info )
call cpftrf( transr, uplo, n, a, info )
call zpftrf( transr, uplo, n, a, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, a Hermitian positive-definite matrix *A*:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data	if *uplo*='U'
$A = L*L^T$ for real data, $A = L*L^H$ for complex data	if *uplo*='L'

where *L* is a lower triangular matrix and *U* is upper triangular.

The matrix *A* is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

This is the block version of the algorithm, calling Level 3 BLAS.

Input Parameters

transr	CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data). If *transr* = 'N', the Normal *transr* of RFP *A* is stored. If *transr* = 'T', the Transpose *transr* of RFP *A* is stored. If *transr* = 'C', the Conjugate-Transpose *transr* of RFP *A* is stored.
uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of *A* is stored: If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*. If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
a	REAL for spftrf DOUBLE PRECISION for dpftrf COMPLEX for cpftrf DOUBLE COMPLEX for zpftrf. Array, size (*n**(*n*+1)/2). The array *a* contains the matrix *A* in the RFP format.

Output Parameters

a	*a* is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo* and *trans*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

See Also

Matrix Storage Schemes

?pptrf
Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using packed storage.

Syntax

```
call spptrf( uplo, n, ap, info )
call dpptrf( uplo, n, ap, info )
call cpptrf( uplo, n, ap, info )
call zpptrf( uplo, n, ap, info )
```

```
call pptrf( ap [, uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite packed matrix *A*:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data	if *uplo*='U'
$A = L*L^T$ for real data, $A = L*L^H$ for complex data	if *uplo*='L'

where *L* is a lower triangular matrix and *U* is upper triangular.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is packed in the array *ap*, and how *A* is factored:

If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix *A*, and *A* is factored as U^H*U.

If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix *A*; *A* is factored as $L*L^H$.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

ap

REAL for spptrf

DOUBLE PRECISION for dpptrf

COMPLEX for cpptrf

DOUBLE COMPLEX for zpptrf.

Array, size at least max(1, $n(n+1)/2$). The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in packed storage (see Matrix Storage Schemes).

Output Parameters

ap

Overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.

info

INTEGER. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `pptrf` interface are as follows:

ap	Holds the array *A* of size `(n*(n+1)/2)`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

If *uplo* = `'U'`, the computed factor *U* is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |U^H||U|, |e_{ij}| \leq c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for *uplo* = `'L'`.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

`?pptrs`	to solve *A*X = B*
`?ppcon`	to estimate the condition number of *A*
`?pptri`	to compute the inverse of *A*.

See Also

`mkl_progress`

Matrix Storage Schemes

?pbtrf
Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite band matrix.

Syntax

```
call spbtrf( uplo, n, kd, ab, ldab, info )
call dpbtrf( uplo, n, kd, ab, ldab, info )
call cpbtrf( uplo, n, kd, ab, ldab, info )
call zpbtrf( uplo, n, kd, ab, ldab, info )
call pbtrf( ab [, uplo] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite band matrix *A*:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data if *uplo*='U'

$A = L*L^T$ for real data, $A = L*L^H$ for complex data if *uplo*='L'

where *L* is a lower triangular matrix and *U* is upper triangular.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored in the array *ab*, and how *A* is factored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n — INTEGER. The order of matrix *A*; $n \geq 0$.

kd — INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.

ab — REAL for spbtrf

DOUBLE PRECISION for dpbtrf

COMPLEX for cpbtrf

DOUBLE COMPLEX for zpbtrf.

Array, size (*ldab*,*). The array *ab* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in band storage (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*).

ldab — INTEGER. The leading dimension of the array *ab*. ($ldab \geq kd + 1$)

Output Parameters

ab — The upper or lower triangular part of *A* (in band storage) is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.

info — INTEGER. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `pbtrf` interface are as follows:

ab	Holds the array *A* of size `(kd+1,n)`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

If *uplo* = `'U'`, the computed factor *U* is the exact factor of a perturbed matrix *A* + *E*, where

$$|E| \le c(kd+1)\varepsilon |U^H||U|, |e_{ij}| \le c(kd+1)\varepsilon\sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for *uplo* = `'L'`.

The total number of floating-point operations for real flavors is approximately $n(kd+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that *kd* is much less than *n*.

After calling this routine, you can call the following routines:

`?pbtrs`	to solve *A*X* = *B*
`?pbcon`	to estimate the condition number of *A*.

See Also

`mkl_progress`

Matrix Storage Schemes

?pttrf
Computes the factorization of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

```
call spttrf( n, d, e, info )
call dpttrf( n, d, e, info )
call cpttrf( n, d, e, info )
call zpttrf( n, d, e, info )
call pttrf( d, e [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine forms the factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite tridiagonal matrix *A*:

$A = L*D*L^T$ for real flavors, or

$A = L*D*L^H$ for complex flavors,

where D is diagonal and L is unit lower bidiagonal. The factorization may also be regarded as having the form $A = U^T*D*U$ for real flavors, or $A = U^H*D*U$ for complex flavors, where U is unit upper bidiagonal.

Input Parameters

n	INTEGER. The order of the matrix A; $n \geq 0$.
d	REAL for sptrf, cpttrf DOUBLE PRECISION for dpttrf, zpttrf. Array, dimension (n). Contains the diagonal elements of A.
e	REAL for spttrf DOUBLE PRECISION for dpttrf COMPLEX for cpttrf DOUBLE COMPLEX for zpttrf. Array, dimension (n -1). Contains the subdiagonal elements of A.

Output Parameters

d	Overwritten by the n diagonal elements of the diagonal matrix D from the $L*D*L^T$ (for real flavors) or $L*D*L^H$ (for complex flavors) factorization of A.
e	Overwritten by the $(n - 1)$ sub-diagonal elements of the unit bidiagonal factor L or U from the factorization of A.
info	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i-th parameter had an illegal value. If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite; if $i < n$, the factorization could not be completed, while if $i = n$, the factorization was completed, but $d(n) \leq 0$.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pttrf interface are as follows:

d	Holds the vector of length n.
e	Holds the vector of length (n-1).

?sytrf
Computes the Bunch-Kaufman factorization of a symmetric matrix.

Syntax

```
call ssytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call sytrf( a [, uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the factorization of a real/complex symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

if *uplo*='U', $A = U*D*U^T$
if *uplo*='L', $A = L*D*L^T$

where *A* is the input matrix, *U* and *L* are products of permutation and triangular matrices with unit diagonal (upper triangular for *U* and lower triangular for *L*), and *D* is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. *U* and *L* have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of *D*.

NOTE This routine supports the Progress Routine feature. See Progress Routine for details.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*, and *A* is factored as $U*D*U^T$.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*, and *A* is factored as $L*D*L^T$.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

a

REAL for ssytrf

DOUBLE PRECISION for dsytrf

COMPLEX for csytrf

DOUBLE COMPLEX for zsytrf.

Array, size (*lda*,*) . The array *a* contains either the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*).

lda

INTEGER. The leading dimension of a; at least max(1, *n*).

work — Same type as *a*. A workspace array, dimension at least `max(1,lwork)`.

lwork — `INTEGER`. The size of the *work* array `(lwork≥n)`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.

See Application Notes for the suggested value of *lwork*.

Output Parameters

a — The upper or lower triangular part of `a` is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).

work(1) — If `info=0`, on exit `work(1)` contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

ipiv — `INTEGER`.

Array, size at least `max(1, n)`. Contains details of the interchanges and the block structure of *D*. If `ipiv(i) = k >0`, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = `'U'` and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and `i–1`, and (`i–1`)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = `'L'` and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and `i+1`, and (`i+1`)-th row and column of *A* was interchanged with the *m*-th row and column.

info — `INTEGER`. If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, D_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sytrf` interface are as follows:

a — holds the matrix *A* of size (*n*, *n*)

ipiv — holds the vector of length *n*

uplo — must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* are stored in the corresponding columns of the array *a*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* =1...*n* , then all off-diagonal elements of *U* (*L*) are stored explicitly in the corresponding elements of the array *a*.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T$$

c(*n*) is a modest linear function of *n*, and ε is the machine precision. A similar estimate holds for the computed *L* and *D* when *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?sytrs	to solve *A***X* = *B*
?sycon	to estimate the condition number of *A*
?sytri	to compute the inverse of *A*.

If *uplo* = 'U', then *A* = U*D*U', where

```
U = P(n)*U(n)* ... *P(k)*U(k)*...,
```

that is, U is a product of terms P(*k*)*U(*k*), where

- *k* decreases from *n* to 1 in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(*k*).
- P(*k*) is a permutation matrix as defined by *ipiv*(*k*) .
- U(*k*) is a unit upper triangular matrix, such that if the diagonal block D(*k*) is of order *s* (*s* = 1 or 2), then

$$U(k) = \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \begin{matrix} k-s \\ s \\ n-k \end{matrix}$$
$$\begin{matrix} k-s & s & n-k \end{matrix}$$

If *s* = 1, D(*k*) overwrites *A*(*k*,*k*), and *v* overwrites *A*(1:*k*-1,*k*).

If *s* = 2, the upper triangle of D(*k*) overwrites *A*(*k*-1,*k*-1), *A*(*k*-1,*k*) and *A*(*k*,*k*), and *v* overwrites *A*(1:*k*-2,*k* -1:*k*).

If *uplo* = 'L', then *A* = L*D*L', where

```
L = P(1)*L(1)* ... *P(k)*L(k)*...,
```

that is, L is a product of terms P(*k*)*L(*k*), where

- *k* increases from 1 to *n* in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(*k*).
- P(*k*) is a permutation matrix as defined by *ipiv*(*k*).
- L(*k*) is a unit lower triangular matrix, such that if the diagonal block D(*k*) is of order *s* (*s* = 1 or 2), then

$$L(k) = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix} \begin{matrix} k-1 \\ s \\ n-k-s+1 \end{matrix}$$
$$\begin{matrix} k-1 & s & n-k-s+1 \end{matrix}$$

If *s* = 1, D(*k*) overwrites *A*(*k*,*k*), and *v* overwrites *A*(*k*+1:*n*,*k*).

If *s* = 2, the lower triangle of D(*k*) overwrites *A*(*k*,*k*), *A*(*k*+1,*k*), and *A*(*k*+1,*k*+1), and *v* overwrites *A*(*k* +2:*n*,*k*:*k*+1).

See Also

mkl_progress

Matrix Storage Schemes

?sytrf_aa
Computes the factorization of a symmetric matrix using Aasen's algorithm.

```
call ssytrf_aa(uplo, n, A, lda, ipiv, work, lwork, info)
call dsytrf_aa(uplo, n, A, lda, ipiv, work, lwork, info)
call csytrf_aa(uplo, n, A, lda, ipiv, work, lwork, info)
```

```
call zsytrf_aa(uplo, n, A, lda, ipiv, work, lwork, info)
```

Description

?sytrf_aa computes the factorization of a symmetric matrix A using Aasen's algorithm. The form of the factorization is A = U*T*U^T or A = L*T*L^T where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is a complex symmetric tridiagonal matrix.

This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

uplo CHARACTER*1

- = 'U': The upper triangle of A is stored.
- = 'L': The lower triangle of A is stored.

n INTEGER

The order of the matrix A. *n* ≥ 0.

A REAL for ssytrf_aa

DOUBLE PRECISION for dsytrf_aa

COMPLEX for csytrf_aa

COMPLEX*16 for zsytrf_aa

Array, dimension (*lda*,*n*). On entry, the symmetric matrix A. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

lda INTEGER

The leading dimension of the array *A*. *lda* ≥ max(1, *n*).

lwork INTEGER

The length of the array *work*.

If *lwork* = -1, a workspace query is assumed; the routine calculates only the optimal size of the *work* array and returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by XERBLA.

Output Parameters

A REAL for ssytrf_aa

DOUBLE PRECISION for dsytrf_aa

COMPLEX for csytrf_aa

COMPLEX*16 for zsytrf_aa

On exit, the tridiagonal matrix is stored in the diagonals and the subdiagonals of A just below (or above) the diagonals, and L is stored below (or above) the subdiagonals, when *uplo* is 'L' (or 'U').

ipiv INTEGER

Array, dimension (*n*). On exit, it contains the details of the interchanges; that is, the row and column *k* of A were interchanged with the row and column *ipiv*(*k*).

work REAL for ssytrf_aa

DOUBLE PRECISION for dsytrf_aa

COMPLEX for csytrf_aa

COMPLEX*16 for zsytrf_aa

Array, dimension (max(1, *lwork*)). On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info INTEGER

- = 0: successful exit.
- < 0: If *info* = -*i*, the i^{th} argument had an illegal value.
- > 0: If *info* = *i*, D(*i*,*i*) is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?sytrf_rook

Computes the bounded Bunch-Kaufman factorization of a symmetric matrix.

Syntax

```
call ssytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call sytrf_rook( a [, uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the factorization of a real/complex symmetric matrix *A* using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The form of the factorization is:

if *uplo*='U', $A = U*D*U^T$
if *uplo*='L', $A = L*D*L^T$,

where *A* is the input matrix, *U* and *L* are products of permutation and triangular matrices with unit diagonal (upper triangular for *U* and lower triangular for *L*), and *D* is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. *U* and *L* have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of *D*.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*, and *A* is factored as $U*D*U^T$.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*, and *A* is factored as $L*D*L^T$.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

a

REAL for ssytrf_rook

DOUBLE PRECISION for dsytrf_rook

COMPLEX for csytrf_rook

DOUBLE COMPLEX for zsytrf_rook.

Array, size (*lda*, *n*). The array *a* contains either the upper or the lower triangular part of the matrix *A* (see *uplo*).

lda

INTEGER. The leading dimension of *a*; at least max(1, *n*).

work

Same type as *a*. A workspace array, dimension at least max(1, *lwork*).

lwork

INTEGER. The size of the *work* array (*lwork*≥*n*).

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See ?sytrf Application Notes for the suggested value of *lwork*.

Output Parameters

a

The upper or lower triangular part of *a* is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).

work(1)

If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

ipiv

INTEGER.

Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D*.

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) < 0 and *ipiv*(*k* − 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* - 1 and -*ipiv*(*k* - 1) were interchanged, and $D_{k-1:k,\ k-1:k}$ is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) < 0 and *ipiv*(*k* + 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* + 1 and -*ipiv*(*k* + 1) were interchanged, and $D_{k:k+1,\ k:k+1}$ is a 2-by-2 diagonal block.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, D_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sytrf_rook interface are as follows:

a — holds the matrix *A* of size (*n*, *n*)

ipiv — holds the vector of length *n*

uplo — must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?sytrs_rook — to solve *A*X* = *B*

?sycon_rook — to estimate the condition number of *A*

?sytri_rook — to compute the inverse of *A*.

If *uplo* = 'U', then *A* = U*D*U', where

```
U = P(n)*U(n)* ... *P(k)*U(k)*...,
```

that is, U is a product of terms P(*k*)*U(*k*), where

- *k* decreases from *n* to 1 in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(*k*).
- P(*k*) is a permutation matrix as defined by *ipiv*(*k*).
- U(*k*) is a unit upper triangular matrix, such that if the diagonal block D(*k*) is of order *s* (*s* = 1 or 2), then

$$U(k) = \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \begin{matrix} k-s \\ s \\ n-k \end{matrix}$$
$$\begin{matrix} k-s & s & n-k \end{matrix}$$

If *s* = 1, D(*k*) overwrites *A*(*k*,*k*), and *v* overwrites *A*(1:*k*-1,*k*).

If *s* = 2, the upper triangle of D(*k*) overwrites *A*(*k*-1,*k*-1), *A*(*k*-1,*k*) and *A*(*k*,*k*), and *v* overwrites *A*(1:*k*-2,*k* -1:*k*).

If *uplo* = 'L', then *A* = L*D*L', where

```
L = P(1)*L(1)* ... *P(k)*L(k)*...,
```

that is, L is a product of terms P(*k*)*L(*k*), where

- *k* increases from 1 to *n* in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks D(*k*).
- P(*k*) is a permutation matrix as defined by *ipiv*(*k*).
- L(*k*) is a unit lower triangular matrix, such that if the diagonal block D(*k*) is of order *s* (*s* = 1 or 2), then

$$L(k) = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix} \begin{matrix} k-1 \\ s \\ n-k-s+1 \end{matrix}$$
$$\begin{matrix} k-1 & s & n-k-s+1 \end{matrix}$$

If *s* = 1, D(*k*) overwrites *A*(*k*,*k*), and *v* overwrites *A*(*k*+1:*n*,*k*).

If *s* = 2, the lower triangle of D(*k*) overwrites *A*(*k*,*k*), *A*(*k*+1,*k*), and *A*(*k*+1,*k*+1), and *v* overwrites *A*(*k* +2:*n*,*k*:*k*+1).

See Also

Matrix Storage Schemes

?sytrf_rk
Computes the factorization of a real or complex symmetric indefinite matrix using the bounded Bunch-Kaufman (rook) diagonal pivoting method (BLAS3 blocked algorithm).

```
call ssytrf_rk(uplo, n, A, lda, e, ipiv, work, lwork, info)
call dsytrf_rk(uplo, n, A, lda, e, ipiv, work, lwork, info)
call csytrf_rk(uplo, n, A, lda, e, ipiv, work, lwork, info)
```

```
call zsytrf_rk(uplo, n, A, lda, e, ipiv, work, lwork, info)
```

Description

?sytrf_rk computes the factorization of a real or complex symmetric matrix A using the bounded Bunch-Kaufman (rook) diagonal pivoting method: A= P*U*D*(U^T)*(P^T) or A = P*L*D*(L^T)*(P^T), where U (or L) is unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level-3 BLAS.

Input Parameters

uplo

CHARACTER*1

Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

- = 'U': Upper triangular
- = 'L': Lower triangular

n

INTEGER

The order of the matrix A. *n* ≥ 0.

A

REAL for ssytrf_rk

DOUBLE PRECISION for dsytrf_rk

COMPLEX for csytrf_rk

COMPLEX*16 for zsytrf_rk

Array, dimension (*lda*,*n*). On entry, the symmetric matrix A. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

lda

INTEGER

The leading dimension of the array *A*.*lda* ≥ max(1, *n*).

lwork

INTEGER

The length of the array *work*.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by XERBLA.

Output Parameters

A

REAL for ssytrf_rk

DOUBLE PRECISION for dsytrf_rk

COMPLEX for csytrf_rk

COMPLEX*16 for zsytrf_rk

On exit, contains:

- *Only* diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, D(*k*,*k*) = A(*k*,*k*); (superdiagonal (or subdiagonal) elements of D are stored on exit in array *e*).
- If *uplo* = 'U', factor U in the superdiagonal part of A. If *uplo* = 'L', factor L in the subdiagonal part of A.

e

REAL for ssytrf_rk

DOUBLE PRECISION for dsytrf_rk

COMPLEX for csytrf_rk

COMPLEX*16 for zsytrf_rk

Array, dimension (*n*). On exit, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', e(*i*) = D(*i*–1,*i*), *i*=2:N, and e(1) is set to 0. If *uplo* = 'L', e(*i*) = D(*i*+1,*i*), *i*=1:N–1, and e(*n*) is set to 0.

NOTE For 1-by-1 diagonal block D(*k*), where 1 ≤ *k* ≤ *n*, the element *e*(*k*) is set to 0 in both the *uplo* = 'U' and *uplo* = 'L' cases.

ipiv

INTEGER

Array, dimension (*n*).*ipiv* describes the permutation matrix P in the factorization of matrix A as follows: The absolute value of ipiv(*k*) represents the index of the row and column that were interchanged with the k^{th} row and column. The value of *uplo* describes the order in which the interchanges were applied. Also, the sign of *ipiv* represents the block structure of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks, which correspond to 1 or 2 interchanges at each factorization step. If *uplo* = 'U' (in factorization order, *k* decreases from *n* to 1):

1. A single positive entry ipiv(*k*) > 0 means that D(*k*,*k*) is a 1-by-1 diagonal block. If ipiv(*k*) != *k*, rows and columns *k* and ipiv(*k*) were interchanged in the matrix A(1:N,1:N). If ipiv(*k*) = *k*, no interchange occurred.
2. A pair of consecutive negative entries ipiv(*k*) < 0 and ipiv(*k*–1). < 0 means that D(*k*–1:*k*,*k*–1:*k*) is a 2-by-2 diagonal block. (Note that negative entries in *ipiv* appear *only* in pairs.)
   - If –ipiv(*k*) != *k*, rows and columns *k* and –ipiv(*k*) were interchanged in the matrix A(1:N,1:N). If –ipiv(*k*) = *k*, no interchange occurred.
   - If –ipiv(*k*–1) != *k*–1, rows and columns *k*–1 and –ipiv(*k*–1) were interchanged in the matrix A(1:N,1:N). If –ipiv(*k*–1) = *k*–1, no interchange occurred.
3. In both cases 1 and 2, always ABS(ipiv(*k*)) ≤ *k*.

NOTE Any entry ipiv(*k*) is always nonzero on output.

If *uplo* = 'L' (in factorization order, *k* increases from 1 to *n*):

1. A single positive entry ipiv(*k*) > 0 means that D(*k*,*k*) is a 1-by-1 diagonal block. If ipiv(*k*) != *k*, rows and columns *k* and ipiv(*k*) were interchanged in the matrix A(1:N,1:N). If ipiv(*k*) = *k*, no interchange occurred.
2. A pair of consecutive negative entries ipiv(*k*) < 0 and ipiv(*k*+1) < 0 means that D(*k*:*k*+1,*k*:*k*+1) is a 2-by-2 diagonal block. (Note that negative entries in *ipiv* appear *only* in pairs.)
   - If –ipiv(*k*) != *k*, rows and columns *k* and –ipiv(*k*) were interchanged in the matrix A(1:N,1:N). If –ipiv(*k*) = *k*, no interchange occurred.
   - If –ipiv(*k*+1) != *k*+1, rows and columns *k*–1 and –ipiv(*k*–1) were interchanged in the matrix A(1:N,1:N). If –ipiv(*k*+1) = *k*+1, no interchange occurred.
3. In both cases 1 and 2, always ABS(ipiv(*k*)) ≥ *k*.

NOTE Any entry ipiv(*k*) is always nonzero on output.

work

`REAL` for `ssytrf_rk`

`DOUBLE PRECISION` for `dsytrf_rk`

`COMPLEX` for `csytrf_rk`

`COMPLEX*16` for `zsytrf_rk`

Array, dimension (MAX(1,*lwork*)). On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info

`INTEGER`

- = 0: Successful exit.
- < 0: If *info* = –*k*, the k^{th} argument had an illegal value.
- > 0: If *info* = *k*, the matrix A is singular. If *uplo* = `'U'`, column *k* in the upper triangular part of A contains all zeros. If *uplo* = `'L'`, column *k* in the lower triangular part of A contains all zeros. Therefore, D(*k*,*k*) is exactly zero, and superdiagonal elements of column *k* of U (or subdiagonal elements of column *k* of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

NOTE *info* stores only the first occurrence of a singularity; any subsequent occurrence of singularity is not stored in *info* even though the factorization always completes.

?hetrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

```
call chetrf( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf( uplo, n, a, lda, ipiv, work, lwork, info )
```

```
call hetrf( a [, uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the factorization of a complex Hermitian matrix *A* using the Bunch-Kaufman diagonal pivoting method:

if *uplo*='U', $A = U*D*U^H$
if *uplo*='L', $A = L*D*L^H$,

where *A* is the input matrix, *U* and *L* are products of permutation and triangular matrices with unit diagonal (upper triangular for *U* and lower triangular for *L*), and *D* is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. *U* and *L* have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of *D*.

> **NOTE**
> This routine supports the Progress Routine feature. See Progress Routine for details.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*, and *A* is factored as $U*D*U^H$.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*, and *A* is factored as $L*D*L^H$.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

a, *work*

COMPLEX for chetrf

DOUBLE COMPLEX for zhetrf.

Arrays, size (*lda*,*), *work*(*).

The array *a* contains the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*).

work(*) is a workspace array of dimension at least max(1, *lwork*).

lda

INTEGER. The leading dimension of *a*; at least max(1, *n*).

lwork

INTEGER. The size of the *work* array ($lwork \geq n$).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See Application Notes for the suggested value of *lwork*.

Output Parameters

a	The upper or lower triangular part of *a* is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
ipiv	INTEGER. Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D*. If *ipiv*(*i*) = *k* >0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column. If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column. If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hetrf interface are as follows:

a	holds the matrix *A* of size (*n*, *n*)
ipiv	holds the vector of length *n*
uplo	must be 'U' or 'L'. The default value is 'U'.

Application Notes

This routine is suitable for Hermitian matrices that are not known to be positive-definite. If *A* is in fact positive-definite, the routine does not perform interchanges, and no 2-by-2 diagonal blocks occur in *D*.

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* are stored in the corresponding columns of the array *a*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If*ipiv*(*i*) = *i* for all $i = 1 \ldots n$, then all off-diagonal elements of *U* (*L*) are stored explicitly in the corresponding elements of the array *a*.

If `uplo = 'U'`, the computed factors *U* and *D* are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for the computed *L* and *D* when `uplo = 'L'`.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

`?hetrs`	to solve $A*X = B$
`?hecon`	to estimate the condition number of *A*
`?hetri`	to compute the inverse of *A*.

See Also

`mkl_progress`

Matrix Storage Schemes

?hetrf_aa
Computes the factorization of a complex hermitian matrix using Aasen's algorithm.

```
call chetrf_aa(uplo, n, a, lda, ipiv, work, lwork, info)
call zhetrf_aa(uplo, n, a, lda, ipiv, work, lwork, info)
```

Description

`?hetrf_aa` computes the factorization of a complex Hermitian matrix *A* using Aasen's algorithm. The form of the factorization is $A = U * T * U^H$ or $a = L*T*L^H$ where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is a Hermitian tridiagonal matrix. This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

uplo	`CHARACTER*1`. = 'U': Upper triangle of *A* is stored; = 'L': Lower triangle of *a* is stored.
n	`INTEGER`. The order of the matrix *A*. $n \geq 0$.

a — COMPLEX for chetrf_aa

COMPLEX*16 for zhetrf_aa

Array of size (*lda*, *n*). On entry, the Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*n*).

lwork — INTEGER. The length of *work*. *lwork*≥ 2**n*. For optimum performance *lwork*≥*n**(1 + *nb*), where *nb* is the optimal block size. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

a — On exit, the tridiagonal matrix is stored in the diagonals and the subdiagonals of *a* just below (or above) the diagonals, and *L* is stored below (or above) the subdiagonals, when *uplo* is 'L' (or 'U').

ipiv — INTEGER . array, dimension (*n*) On exit, it contains the details of the interchanges: the row and column *k* of *a* were interchanged with the row and column *ipiv*(*k*).

work — COMPLEX for chetrf_aa

COMPLEX*16 for zhetrf_aa

Array of size (max(1, *lwork*)). On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info — INTEGER.

If *info* = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value,

If *info* > 0: if *info* = *i*, $D_{i,i}$ is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hetrf_rook
Computes the bounded Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

```
call chetrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call hetrf_rook( a [, uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the factorization of a complex Hermitian matrix *A* using the bounded Bunch-Kaufman diagonal pivoting method:

if *uplo*='U', *A* = *U***D***U*^{H}
if *uplo*='L', *A* = *L***D***L*^{H},

where *A* is the input matrix, *U* (or *L*) is a product of permutation and unit upper (or lower) triangular matrices, and *D* is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*.

n

INTEGER. The order of matrix *A*; *n*≥ 0.

a

COMPLEX for chetrf_rook

COMPLEX*16 for zhetrf_rook.

Array *a*, size (*lda*,n).

The array *a* contains the upper or the lower triangular part of the matrix *A* (see *uplo*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda

INTEGER. The leading dimension of *a*; at least max(1, *n*).

work

COMPLEX for chetrf_rook

COMPLEX*16 for zhetrf_rook.

Array *work*(*).

work(*) is a workspace array of dimension at least max(1, *lwork*).

lwork

INTEGER. The size of the *work* array (*lwork*≥*n*).

The length of *work*. *lwork*≥ 1. For best performance *lwork*≥*n***nb*, where *nb* is the block size returned by ilaenv.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

a

The block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* (see Application Notes for further details).

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

ipiv

INTEGER.

Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D*.

- If *uplo* = 'U':

  If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

  If *ipiv*(*k*) < 0 and *ipiv*(*k* - 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged and rows and columns *k* - 1 and -*ipiv*(*k* - 1) were interchanged, $D_{k-1:k,k-1:k}$ is a 2-by-2 diagonal block.
- If *uplo* = 'L':

  If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

  If *ipiv*(*k*) < 0 and *ipiv*(*k* + 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged and rows and columns *k* + 1 and -*ipiv*(*k* + 1) were interchanged, $D_{k:k+1,k:k+1}$ is a 2-by-2 diagonal block.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, D_{ii} is exactly 0. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by 0 will occur if you use *D* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hetrf_rook interface are as follows:

a

holds the matrix *A* of size (*n*, *n*)

ipiv

holds the vector of length *n*

uplo

must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', then$A = U*D*U^H$, where

U = *P*(*n*)**U*(*n*)* ... **P*(*k*)*U*(*k*)* ...,

i.e., *U* is a product of terms *P*(*k*)**U*(*k*), where *k* decreases from *n* to 1 in steps of 1 or 2, and *D* is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks *D*(*k*). *P*(*k*) is a permutation matrix as defined by *ipiv*(*k*), and *U*(*k*) is a unit upper triangular matrix, such that if the diagonal block *D*(*k*) is of order *s* (*s* = 1 or 2), then

$$U(k) = \begin{array}{c} \\ k-s \\ s \\ n-k \end{array} \begin{array}{c} \begin{array}{ccc} k-s & s & n-k \end{array} \\ \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \end{array}$$

If *s* = 1, *D*(*k*) overwrites *A*(*k*,*k*), and *v* overwrites *A*(1:*k*-1,*k*).

If *s* = 2, the upper triangle of *D*(*k*) overwrites *A*(*k*-1,*k*-1), *A*(*k*-1,*k*), and *A*(*k*,*k*), and *v* overwrites *A*(1:*k*-2,*k*-1:*k*).

If *uplo* = 'L', then $A = L*D*L^H$, where

L = *P*(1)**L*(1)* ... **P*(*k*)**L*(*k*)* ...,

i.e., *L* is a product of terms *P*(*k*)**L*(*k*), where *k* increases from 1 to *n* in steps of 1 or 2, and *D* is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks *D*(*k*). *P*(*k*) is a permutation matrix as defined by *ipiv*(*k*), and *L*(*k*) is a unit lower triangular matrix, such that if the diagonal block *D*(*k*) is of order *s* (*s* = 1 or 2), then

$$L(k) = \begin{array}{c} \\ k-1 \\ s \\ n-k-s+1 \end{array} \begin{array}{c} \begin{array}{ccc} k-1 & s & n-k-s+1 \end{array} \\ \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix} \end{array}$$

If *s* = 1, *D*(*k*) overwrites *A*(*k*,*k*), and *v* overwrites *A*(*k*+1:*n*,*k*).

If *s* = 2, the lower triangle of *D*(*k*) overwrites *A*(*k*,*k*), *A*(*k*+1,*k*), and *A*(*k*+1,*k*+1), and *v* overwrites *A*(*k*+2:*n*,*k*:*k*+1).

See Also

mkl_progress

Matrix Storage Schemes

?hetrf_rk
Computes the factorization of a complex Hermitian indefinite matrix using the bounded Bunch-Kaufman (rook) diagonal pivoting method (BLAS3 blocked algorithm).

```
call chetrf_rk(uplo, n, A, lda, e, ipiv, work, lwork, info)
call zhetrf_rk(uplo, n, A, lda, e, ipiv, work, lwork, info)
```

Description

?hetrf_rk computes the factorization of a complex Hermitian matrix A using the bounded Bunch-Kaufman (rook) diagonal pivoting method: A = P*U*D*(U^H)*(P^T) or A = P*L*D*(L^H)*(P^T), where U (or L) is unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

uplo CHARACTER*1

Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

- = 'U': Upper triangular.
- = 'L': Lower triangular.

n INTEGER

The order of the matrix A. *n* ≥ 0.

A COMPLEX for chetrf_rk

COMPLEX*16 for zhetrf_rk

Array, dimension (*lda*,*n*). On entry, the Hermitian matrix A. If *uplo* = 'U': The leading *n*-by-*n* upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If *uplo* = 'L': The leading *n*-by-*n* lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

lda INTEGER

The leading dimension of the array *A*. *lda* ≥ max(1, *n*).

lwork INTEGER

The length of the array *work*.

If *lwork* = –1, a workspace query is assumed; the routine calculates only the optimal size of the *work* array and returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by XERBLA.

Output Parameters

A COMPLEX for chetrf_rk

COMPLEX*16 for zhetrf_rk

On exit, contains:

- *Only* diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, D(*k*,*k*) = A(*k*,*k*). Superdiagonal (or subdiagonal) elements of D are stored on exit in array *e*.

  —and—
- If *uplo* = 'U', factor U in the superdiagonal part of A. If *uplo* = 'L', factor L in the subdiagonal part of A.

e COMPLEX for chetrf_rk

COMPLEX*16 for zhetrf_rk

Array, dimension (*n*). On exit, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', e(*i*) = D(*i*–1,*i*), *i*=2:N, and e(1) is set to 0. If *uplo* = 'L', e(*i*) = D(*i*+1,*i*), *i*=1:N–1, and e(*n*) is set to 0.

NOTE For 1-by-1 diagonal block D(*k*), where 1 ≤ k ≤ *n*, the element *e*(*k*) is set to 0 in both the *uplo* = 'U' and *uplo* = 'L' cases.

ipiv INTEGER

Array, dimension (*n*). *ipiv* describes the permutation matrix P in the factorization of matrix A as follows. The absolute value of *ipiv*(*k*) represents the index of row and column that were interchanged with the k^{th} row and column. The value of *uplo* describes the order in which the interchanges were applied. Also, the sign of *ipiv* represents the block structure of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks that correspond to 1 or 2 interchanges at each factorization step. If *uplo* = 'U' (in factorization order, *k* decreases from *n* to 1):

1. A single positive entry ipiv(*k*) > 0 means that D(*k*,*k*) is a 1-by-1 diagonal block. If ipiv(*k*) != *k*, rows and columns *k* and ipiv(*k*) were interchanged in the matrix A(1:N,1:N). If ipiv(*k*) = *k*, no interchange occurred.
2. A pair of consecutive negative entries ipiv(*k*) < 0 and ipiv(*k*–1) < 0 means that D(*k*–1:*k*,*k*–1:*k*) is a 2-by-2 diagonal block. (Note that negative entries in *ipiv* appear *only* in pairs.)
   - If –ipiv(*k*) != *k*, rows and columns *k* and –ipiv(*k*) were interchanged in the matrix A(1:N,1:N). If –ipiv(*k*) = *k*, no interchange occurred.
   - If –ipiv(*k*–1) != *k*–1, rows and columns *k*–1 and –ipiv(*k*–1) were interchanged in the matrix A(1:N,1:N). If –ipiv(*k*–1) = *k*–1, no interchange occurred.
3. In both cases 1 and 2, always ABS(ipiv(*k*)) ≤ *k*.

NOTE Any entry ipiv(*k*) is always nonzero on output.

If *uplo* = 'L' (in factorization order, *k* increases from 1 to *n*):

1. A single positive entry ipiv(*k*) > 0 means that D(*k*,*k*) is a 1-by-1 diagonal block. If ipiv(*k*) != *k*, rows and columns *k* and ipiv(*k*) were interchanged in the matrix A(1:N,1:N). If ipiv(*k*) = *k*, no interchange occurred.
2. A pair of consecutive negative entries ipiv(*k*) < 0 and ipiv(*k*+1) < 0 means that D(*k*:*k*+1,*k*:*k*+1) is a 2-by-2 diagonal block. (Note that negative entries in *ipiv* appear *only* in pairs.)
   - If –ipiv(*k*) != *k*, rows and columns *k* and –ipiv(*k*) were interchanged in the matrix A(1:N,1:N). If –ipiv(*k*) = *k*, no interchange occurred.
   - If –ipiv(*k*+1) != *k*+1, rows and columns *k*–1 and –ipiv(*k*–1) were interchanged in the matrix A(1:N,1:N). If –ipiv(*k*+1) = *k*+1, no interchange occurred.
3. In both cases 1 and 2, always ABS(ipiv(*k*)) ≥ *k*.

NOTE Any entry ipiv(*k*) is always nonzero on output.

work

COMPLEX for chetrf_rk

COMPLEX*16 for zhetrf_rk

Array, dimension (MAX(1,*lwork*)). On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info

INTEGER

- = 0: Successful exit.
- < 0: If *info* = -*k*, the k^{th} argument had an illegal value.
- > 0: If *info* = *k*, the matrix A is singular. If *uplo* = 'U', the column *k* in the upper triangular part of A contains all zeros. If *uplo* = 'L', the column *k* in the lower triangular part of A contains all zeros. Therefore D(*k*,*k*) is exactly zero, and superdiagonal elements of column *k* of U (or subdiagonal elements of column *k* of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

NOTE *info* stores only the first occurrence of a singularity; any subsequent occurrence of a singularity is not stored in *info* even though the factorization always completes.

?sptrf

Computes the Bunch-Kaufman factorization of a symmetric matrix using packed storage.

Syntax

```
call ssptrf( uplo, n, ap, ipiv, info )
call dsptrf( uplo, n, ap, ipiv, info )
call csptrf( uplo, n, ap, ipiv, info )
call zsptrf( uplo, n, ap, ipiv, info )
call sptrf( ap [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the factorization of a real/complex symmetric matrix *A* stored in the packed format using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

if *uplo*='U', $A = U*D*U^T$
if *uplo*='L', $A = L*D*L^T$,

where *U* and *L* are products of permutation and triangular matrices with unit diagonal (upper triangular for *U* and lower triangular for *L*), and *D* is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. *U* and *L* have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of *D*.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates whether the upper or lower triangular part of *A* is packed in the array *ap* and how *A* is factored:

If *uplo* = `'U'`, the array *ap* stores the upper triangular part of the matrix *A*, and *A* is factored as $U*D*U^T$.

If *uplo* = `'L'`, the array *ap* stores the lower triangular part of the matrix *A*, and *A* is factored as $L*D*L^T$.

n

`INTEGER`. The order of matrix *A*; $n \geq 0$.

ap

`REAL` for `ssptrf`

`DOUBLE PRECISION` for `dsptrf`

`COMPLEX` for `csptrf`

`DOUBLE COMPLEX` for `zsptrf`.

Array, size at least max(1, *n*(*n*+1)/2). The array *ap* contains the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *packed storage* (see Matrix Storage Schemes).

Output Parameters

ap

The upper or lower triangle of *A* (as specified by *uplo*) is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).

ipiv

`INTEGER`.

Array, size at least `max(1, n)`. Contains details of the interchanges and the block structure of *D*. If *ipiv*(*i*) = *k* >0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = `'U'` and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = `'L'` and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

info

`INTEGER`. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sptrf` interface are as follows:

ap	Holds the array *A* of size `(n*(n+1)/2)`.
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* overwrite elements of the corresponding columns of the array *ap*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* = 1...*n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in packed form.

If *uplo* = `'U'`, the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision. A similar estimate holds for the computed *L* and *D* when *uplo* = `'L'`.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

`?sptrs`	to solve *A*X* = *B*
`?spcon`	to estimate the condition number of *A*
`?sptri`	to compute the inverse of *A*.

See Also

`mkl_progress`

Matrix Storage Schemes

?hptrf
Computes the Bunch-Kaufman factorization of a complex Hermitian matrix using packed storage.

Syntax

```
call chptrf( uplo, n, ap, ipiv, info )
call zhptrf( uplo, n, ap, ipiv, info )
call hptrf( ap [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the factorization of a complex Hermitian packed matrix *A* using the Bunch-Kaufman diagonal pivoting method:

if *uplo*='U', *A* = *U***D***U*^{H}
if *uplo*='L', *A* = *L***D***L*^{H},

where *A* is the input matrix, *U* and *L* are products of permutation and triangular matrices with unit diagonal (upper triangular for *U* and lower triangular for *L*), and *D* is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. *U* and *L* have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of *D*.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is packed and how *A* is factored:

If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix *A*, and *A* is factored as *U***D***U*^{H}.

If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix *A*, and *A* is factored as *L***D***L*^{H}.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

ap

COMPLEX for chptrf

DOUBLE COMPLEX for zhptrf.

Array, size at least max(1, *n*(*n*+1)/2). The array *ap* contains the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *packed storage* (see Matrix Storage Schemes).

Output Parameters

ap

The upper or lower triangle of *A* (as specified by *uplo*) is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).

ipiv

INTEGER.

Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D*. If *ipiv*(*i*) = *k* >0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hptrf interface are as follows:

ap — Holds the array *A* of size (*n**(*n*+1)/2).

ipiv — Holds the vector of length *n*.

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* are stored in the array *ap*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* = 1...*n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in the corresponding elements of the array *a*.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T$$

c(*n*) is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for the computed *L* and *D* when *uplo* = 'L'.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

?hptrs — to solve *A***X* = *B*

?hpcon — to estimate the condition number of *A*

?hptri — to compute the inverse of *A*.

See Also

mkl_progress

Matrix Storage Schemes

mkl_?spffrt2, mkl_?spffrtx
Computes the partial LDL^T factorization of a symmetric matrix using packed storage.

Syntax

```
call mkl_sspffrt2( ap, n, ncolm, work, work2 )
call mkl_dspffrt2( ap, n, ncolm, work, work2 )
call mkl_cspffrt2( ap, n, ncolm, work, work2 )
call mkl_zspffrt2( ap, n, ncolm, work, work2 )
call mkl_sspffrtx( ap, n, ncolm, work, work2 )
call mkl_dspffrtx( ap, n, ncolm, work, work2 )
call mkl_cspffrtx( ap, n, ncolm, work, work2 )
call mkl_zspffrtx( ap, n, ncolm, work, work2 )
```

Include Files

- `mkl.fi`

Description

The routine computes the partial factorization $A = LDL^T$, where L is a lower triangular matrix and D is a diagonal matrix.

> **Caution**
> The routine assumes that the matrix A is factorizable. The routine does not perform pivoting and does not handle diagonal elements which are zero, which cause the routine to produce incorrect results without any indication.

Consider the matrix $A = \begin{pmatrix} a & b^{\mathrm{T}} \\ b & C \end{pmatrix}$, where a is the element in the first row and first column of A, b is a column vector of size n - 1 containing the elements from the second through n-th column of A, C is the lower-right square submatrix of A, and I is the identity matrix.

The `mkl_?spffrt2` routine performs `ncolm` successive factorizations of the form

$$A = \begin{pmatrix} a & b^{\mathrm{T}} \\ b & C \end{pmatrix} = \begin{pmatrix} a & 0 \\ b & I \end{pmatrix} \begin{pmatrix} a^{-1} & 0 \\ 0 & C - ba^{-1}b^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} a & b^{\mathrm{T}} \\ 0 & I \end{pmatrix}.$$

The `mkl_?spffrtx` routine performs `ncolm` successive factorizations of the form

$$A = \begin{pmatrix} a & b^{\mathrm{T}} \\ b & C \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ ba^{-1} & I \end{pmatrix} \begin{pmatrix} a & 0 \\ 0 & C - ba^{-1}b^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} 1 & \left(ba^{-1}\right)^{\mathrm{T}} \\ 0 & I \end{pmatrix}.$$

The approximate number of floating point operations performed by real flavors of these routines is (1/6)*`ncolm`*(2*`ncolm`^{2} - 6*`ncolm`*`n` + 3*`ncolm` + 6*`n`^{2} - 6*`n` + 7).

The approximate number of floating point operations performed by complex flavors of these routines is (1/3)*`ncolm`*(4*`ncolm`^{2} - 12*`ncolm`*`n` + 9*`ncolm` + 12*`n`^{2} - 18*`n` + 8).

Input Parameters

ap	`REAL` for `mkl_sspffrt2` and `mkl_sspffrtx`
	`DOUBLE PRECISION` for `mkl_dspffrt2` and `mkl_dspffrtx`
	`COMPLEX` for `mkl_cspffrt2` and `mkl_cspffrtx`
	`DOUBLE COMPLEX` for `mkl_zspffrt2` and `mkl_zspffrtx`.
	Array, size at least max(1, $n(n+1)/2$). The array *ap* contains the lower triangular part of the matrix *A* in packed storage (see Matrix Storage Schemes for *uplo* = `'L'`).

n	`INTEGER`. The order of matrix *A*; $n \geq 0$.
ncolm	`INTEGER`. The number of columns to factor, $ncolm \leq n$.

work, *work2*	`REAL` for `mkl_sspffrt2` and `mkl_sspffrtx`
	`DOUBLE PRECISION` for `mkl_dspffrt2` and `mkl_dspffrtx`
	`COMPLEX` for `mkl_cspffrt2` and `mkl_cspffrtx`
	`DOUBLE COMPLEX` for `mkl_zspffrt2` and `mkl_zspffrtx`.
	Workspace arrays, size of each at least *n*.

Output Parameters

ap	Overwritten by the factor *L*. The first *ncolm* diagonal elements of the input matrix *A* are replaced with the diagonal elements of *D*. The subdiagonal elements of the first *ncolm* columns are replaced with the corresponding elements of *L*. The rest of the input array is updated as indicated in the Description section.

NOTE
Specifying *ncolm* = *n* results in complete factorization $A = LDL^T$.

See Also

`mkl_progress`

Matrix Storage Schemes

Solving Systems of Linear Equations: LAPACK Computational Routines

This section describes the LAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see Routines for Matrix Factorization). However, the factorization is not necessary if your system of equations has a triangular matrix.

?getrs
Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.

Syntax

```
call sgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call dgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call cgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call zgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call getrs( a, ipiv, b [, trans] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the following systems of linear equations:

$A*X = B$	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Before calling this routine, you must call ?getrf to compute the *LU* factorization of *A*.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', then $A*X = B$ is solved for *X*. If *trans* = 'T', then $A^T*X = B$ is solved for *X*. If *trans* = 'C', then $A^H*X = B$ is solved for *X*.
n	INTEGER. The order of *A*; the number of rows in $B(n \geq 0)$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
a, *b*	REAL for sgetrs DOUBLE PRECISION for dgetrs COMPLEX for cgetrs DOUBLE COMPLEX for zgetrs. Arrays: *a* (size *lda* by *), *b* (size *ldb* by *). The array *a* contains *LU* factorization of matrix *A* resulting from the call of ?getrf. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *a* must be at least max(1,*n*) and the second dimension of *b* at least max(1,*nrhs*).
lda	INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?getrf.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine getrs interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length *n*.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|L||U|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon$$

where $cond(A,x) = \| |A^{-1}||A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $cond(A,x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?gecon.

To refine the solution and estimate the error, call ?gerfs.

See Also

Matrix Storage Schemes

?getrs_batch_strided

Solves a group of systems of linear equations, each with an LU-factored square coefficient matrix and multiple right hand sides.

Syntax

```
call sgetrs_batch_strided(trans, n, nrhs, A, lda, stride_a, ipiv, stride_ipiv, b, ldb, stride_b, batch_size, info)
call dgetrs_batch_strided(trans, n, nrhs, A, lda, stride_a, ipiv, stride_ipiv, b, ldb, stride_b, batch_size, info)
call cgetrs_batch_strided(trans, n, nrhs, A, lda, stride_a, ipiv, stride_ipiv, b, ldb, stride_b, batch_size, info)
call zgetrs_batch_strided(trans, n, nrhs, A, lda, stride_a, ipiv, stride_ipiv, b, ldb, stride_b, batch_size, info)
```

Include Files

```
mkl.fi
```

Description

The `?getrs_batch_strided` routines are similar to the `?getrs` counterparts, but instead solve a group of systems of linear equations. Before calling this routine, you must call `?getrf_batch_strided` to compute the LU factorization of the square coefficient matrix of each linear system.

All coefficient matrices, A_i have the same parameters (matrix size, leading dimension) and are stored at constant *stride_a* from each other. Similarly, all right-hand-side matrices, B_i, have the same parameters and are stored at constant *stride_b* from each other. The respective pivot array associated with each of the LU-factored A_i matrices are stored at constant *stride_ipiv* from each other. The operation is defined as

```
for i = 0 … batch_size-1
    Ai, Bi are matrices at offset i * stride_a, i * stride_b from A and B
    ipivi is an array at offset i * stride_ipiv from ipiv
    Solve the system
    Ai * Xi = Bi   , if trans='N'
    Ai^T * Xi = Bi   , if trans='T'
    Ai^C * Xi = Bi , if trans='C' (for complex matrices)
end for
```

Input Parameters

trans

`CHARACTER*1`. Must be 'N', 'T', or 'C'.

Indicates the form of the systems of linear equations:

If trans = 'N', $A_i * X_i = B_i$ is solved for X_i.

If trans = 'T', $A_i^T * X_i = B_i$ is solved for X_i.

If trans = 'C', $A_i^C * X_i = B_i$ is solved for X_i.

n — INTEGER. The order fo the A_i matrices; $n \geq 0$.

nrhs — INTEGER. The number of right hand sides in each linear system of equations (*nrhs* ≥ 0).

A, B — REAL for sgetrs_batch_strided

DOUBLE PRECISION for dgetrs_batch_strided

COMPLEX for cgetrs_batch_strided

DOUBLE COMPLEX for zgetrs_batch_strided

The *A* array of size at least *stride_a* * *batch_size* holding the LU-factorized A_i matrices resulting from the call to ?getrf_batch_strided.

The *B* array of size at least *stride_b* * *batch_size* holding the B_i matrices, whose columns are the right -hand sides for each linear system of equations.

lda — INTEGER. Specifies the leading dimension of the A_i matrices; *lda* ≥ max(1,*n*).

ldb — INTEGER. Specifies the leading dimension of the B_i matrices; *ldb* ≥ max(1,*n*).

stride_a — INTEGER.

Stride between two consecutive A_i matrices; *stride_a* ≥ *lda* * *n*.

stride_b — INTEGER.

Stride between two consecutive B_i matrices; *stride_b* ≥ *ldb* * *nrhs*.

ipiv — INTEGER.

Array of size at least *stride_ipiv* * *batch_size* holding the pivoting indices for each LU-factorized matrix A_i.

stride_ipiv — INTEGER.

Stride between two consecutive pivot arrays; *stride_ipiv* ≥ *n*).

batch_size — INTEGER.

Number of linear systems to be solved; *batch_size* ≥ 0.

Output Parameters

B — Array is overwritten by the solution matrices X_i.

info — INTEGER.

Array of size at least *batch_size*, which reports the status for each linear system solve.

If *info(i)* = 0, the execution is successful for A_i.

If *info(i)* = -*j*, the *j*-th parameter had an illegal value for A_i.

?getrsnp_batch_strided
Solves a group of systems of linear equations, each with an LU-factored square coefficient matrix and multiple right hand sides.

Syntax

```
call sgetrsnp_batch_strided(trans, n, nrhs, A, lda, stride_a, b, ldb, stride_b, batch_size, info)
call dgetrsnp_batch_strided(trans, n, nrhs, A, lda, stride_a, b, ldb, stride_b, batch_size, info)
call cgetrsnp_batch_strided(trans, n, nrhs, A, lda, stride_a, b, ldb, stride_b, batch_size, info)
call zgetrsnp_batch_strided(trans, n, nrhs, A, lda, stride_a, b, ldb, stride_b, batch_size, info)
```

Include Files

```
mkl.fi
```

Description

The `?getrsnp_batch_strided` routines solve a group of systems of linear equations. Before calling this routine, you must call `?getrfnp_batch_strided` to compute the LU factorization (without pivoting) of the square coefficient matrix of each linear system.

All coefficient matrices, A_i have the same parameters (matrix size, leading dimension) and are stored at constant *stride_a* from each other. Similarly, all right-hand-side matrices, B_i, have the same parameters and are stored at constant *stride_b* from each other. The operation is defined as

```
for i = 0 … batch_size-1
    A_i, B_i are matrices at offset i * stride_a, i * stride_b from A and B
    Solve the system
    A_i * X_i = B_i  , if trans='N'
    A_i^T * X_i = B_i  , if trans='T'
    A_i^C * X_i = B_i , if trans='C' (for complex matrices)
end for
```

Input Parameters

trans
`CHARACTER*1`. Must be 'N', 'T', or 'C'.
Indicates the form of the systems of linear equations:
If trans = 'N', $A_i * X_i = B_i$ is solved for X_i.
If trans = 'T', $A_i^T * X_i = B_i$ is solved for X_i.
If trans = 'C', $A_i^C * X_i = B_i$ is solved for X_i.

n
`INTEGER`. The order fo the A_i matrices; $n \geq 0$.

nrhs
`INTEGER`. The number of right hand sides in each linear system of equations (*nrhs* ≥ 0).

A, B	REAL for sgetrsnp_batch_strided DOUBLE PRECISION for dgetrsnp_batch_strided COMPLEX for cgetrsnp_batch_strided DOUBLE COMPLEX for zgetrsnp_batch_strided The *A* array of size at least *stride_a* * *batch_size* holding the LU-factorized A_i matrices resulting from the call to ?getrfnp_batch_strided. The *B* array of size at least *stride_b* * *batch_size* holding the B_i matrices, whose columns are the right -hand sides for each linear system of equations.
lda	INTEGER. Specifies the leading dimension of the A_i matrices; *lda* ≥ max(1,*n*).
ldb	INTEGER. Specifies the leading dimension of the B_i matrices; *ldb* ≥ max(1,*n*).
stride_a	INTEGER. Stride between two consecutive A_i matrices; *stride_a* ≥ *lda* * *n*.
stride_b	INTEGER. Stride between two consecutive B_i matrices; *stride_b* ≥ *ldb* * *nrhs*.
ipiv	INTEGER. Array of size at least *stride_ipiv* * *batch_size* holding the pivoting indices for each LU-factorized matrix A_i.
batch_size	INTEGER. Number of linear systems to be solved; *batch_size* ≥ 0.

Output Parameters

B	Array is overwritten by the solution matrices X_i.
info	INTEGER. Array of size at least *batch_size*, which reports the status for each linear system solve. If *info(i)* = 0, the execution is successful for A_i. If *info(i)* = -*j*, the *j*-th parameter had an illegal value for A_i.

?gbtrs

Solves a system of linear equations with an LU-factored band coefficient matrix, with multiple right-hand sides.

Syntax

```
call sgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
```

```
call zgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call gbtrs( ab, b, ipiv, [, kl] [, trans] [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the following systems of linear equations:

$A*X = B$	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Here *A* is an *LU*-factored general band matrix of order *n* with *kl* non-zero subdiagonals and *ku* nonzero superdiagonals. Before calling this routine, call ?gbtrf to compute the *LU* factorization of *A*.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'.
n	INTEGER. The order of *A*; the number of rows in *B*; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of *A*; $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of *A*; $ku \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ab, *b*	REAL for sgbtrs DOUBLE PRECISION for dgbtrs COMPLEX for cgbtrs DOUBLE COMPLEX for zgbtrs. Arrays: *ab*(*ldab*,*), *b*(*ldb*,*). The array *ab* contains elements of the LU factors of the matrix *A* as returned by gbtrf. The second dimension of *ab* must be at least max(1, *n*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* at least max(1,*nrhs*).
ldab	INTEGER. The leading dimension of the array *ab*; $ldab \geq 2*kl + ku + 1$.
ldb	INTEGER. The leading dimension of *b*; $ldb \geq$ max(1, *n*).
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?gbtrf.

Output Parameters

b	Overwritten by the solution matrix *X*.

info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gbtrs interface are as follows:

ab	Holds the array *A* of size (2**kl*+*ku*+1, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length min(*m*, *n*).
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *lda*-2**kl*-1.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kl + ku + 1)\varepsilon P|L||U|$$

$c(k)$ is a modest linear function of *k*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kl + ku + 1)\, cond\,(A, x)\varepsilon$$

where $cond(A, x) = \| |A^{-1}||A|\,|x|\,\|_\infty / \|x\|_\infty \leq \||A^{-1}|\|_\infty \||A|\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $2n(ku + 2kl)$ for real flavors. The number of operations for complex flavors is 4 times greater. All these estimates assume that *kl* and *ku* are much less than min(*m*,*n*).

To estimate the condition number $\kappa_\infty(A)$, call ?gbcon.

To refine the solution and estimate the error, call ?gbrfs.

See Also

Matrix Storage Schemes

?gttrs
Solves a system of linear equations with a tridiagonal coefficient matrix using the LU factorization computed by `?gttrf`*.*

Syntax

```
call sgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call dgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call cgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call zgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call gttrs( dl, d, du, du2, b, ipiv [, trans] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine solves for *X* the following systems of linear equations with multiple right hand sides:

$A*X = B$	if *trans*=`'N'`,
$A^T*X = B$	if *trans*=`'T'`,
$A^H*X = B$	if *trans*=`'C'` (for complex matrices only).

Before calling this routine, you must call `?gttrf` to compute the *LU* factorization of *A*.

Input Parameters

trans	`CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`. Indicates the form of the equations: If *trans* = `'N'`, then $A*X = B$ is solved for *X*. If *trans* = `'T'`, then $A^T*X = B$ is solved for *X*. If *trans* = `'C'`, then $A^H*X = B$ is solved for *X*.
n	`INTEGER`. The order of *A*; $n \geq 0$.
nrhs	`INTEGER`. The number of right-hand sides, that is, the number of columns in *B*; $nrhs \geq 0$.

dl,*d*,*du*,*du2*,*b*	`REAL` for `sgttrs`
	`DOUBLE PRECISION` for `dgttrs`
	`COMPLEX` for `cgttrs`
	`DOUBLE COMPLEX` for `zgttrs`.
	Arrays: *dl*(*n* -1), *d*(*n*), *du*(*n* -1), *du2*(*n* -2), *b*(*ldb*,*nrhs*).
	The array *dl* contains the (*n* – 1) multipliers that define the matrix *L* from the *LU* factorization of *A*.

	The array *d* contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*. The array *du* contains the (*n* - 1) elements of the first superdiagonal of *U*. The array *du2* contains the (*n* - 2) elements of the second superdiagonal of *U*. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).
ipiv	INTEGER. Array, size (*n*). The *ipiv* array, as returned by ?gttrf.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gttrs interface are as follows:

dl	Holds the vector of length (*n*-1).
d	Holds the vector of length *n*.
du	Holds the vector of length (*n*-1).
du2	Holds the vector of length (*n*-2).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length *n*.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|L||U|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kl + ku + 1)\, cond\,(A, x)\varepsilon$$

where cond(*A*, *x*) = || $|A^{-1}||A|$ $|x|$ $||_\infty$ / $||x||_\infty \leq ||A^{-1}||_\infty ||A||_\infty = \kappa_\infty(A)$.

Note that cond(*A*, *x*) can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is 7*n* (including *n* divisions) for real flavors and 34*n* (including 2*n* divisions) for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?gtcon.

To refine the solution and estimate the error, call ?gtrfs.

See Also

Matrix Storage Schemes

?dttrsb

Solves a system of linear equations with a diagonally dominant tridiagonal coefficient matrix using the LU factorization computed by ?dttrfb.

Syntax

```
call sdttrsb( trans, n, nrhs, dl, d, du, b, ldb, info )
call ddttrsb( trans, n, nrhs, dl, d, du, b, ldb, info )
call cdttrsb( trans, n, nrhs, dl, d, du, b, ldb, info )
call zdttrsb( trans, n, nrhs, dl, d, du, b, ldb, info )
call dttrsb( dl, d, du, b [, trans] [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The ?dttrsb routine solves the following systems of linear equations with multiple right hand sides for *X*:

*A*X = B*	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Before calling this routine, call ?dttrfb to compute the factorization of *A*.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations solved for *X*:

If *trans* = 'N', then $A*X = B$.

If *trans* = 'T', then $A^T*X = B$.

If *trans* = 'C', then $A^H*X = B$.

n — INTEGER. The order of *A*; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides, that is, the number of columns in *B*; $nrhs \geq 0$.

dl, *d*, *du*, *b* — REAL for sdttrsb

DOUBLE PRECISION for ddttrsb

COMPLEX for cdttrsb

DOUBLE COMPLEX for zdttrsb.

Arrays: *dl*(*n* -1), *d*(*n*), *du*(*n* -1), *b*(*ldb*,*nrhs*).

The array *dl* contains the (*n* - 1) multipliers that define the matrices L_1, L_2 from the factorization of *A*.

The array *d* contains the *n* diagonal elements of the upper triangular matrix *U* from the factorization of *A*.

The array *du* contains the (*n* - 1) elements of the superdiagonal of *U*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldb — INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b — Overwritten by the solution matrix *X*.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

Syntax

```
call spotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call dpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call cpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call zpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call potrs( a, b [,uplo] [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations *A*X = B* with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix *A*, given the Cholesky factorization of *A*:

A = U^T**U* for real data, *A = U*^{H}**U* for complex data if *uplo*='U'

*A = L*L*^{T} for real data, *A = L*L*^{H} for complex data if *uplo*='L'

where *L* is a lower triangular matrix and *U* is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*.

Before calling this routine, you must call ?potrf to compute the Cholesky factorization of *A*.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo = 'U'*, *U* is stored, where*A* = U^T**U* for real data, *A* = U^H**U* for complex data.

If *uplo = 'L'*, *L* is stored, where*A* = *L**L^T for real data, *A* = *L**L^H for complex data.

n INTEGER. The order of matrix *A*; *n*≥ 0.

nrhs INTEGER. The number of right-hand sides (*nrhs*≥ 0).

a, *b* REAL for spotrs

DOUBLE PRECISION for dpotrs

COMPLEX for cpotrs

DOUBLE COMPLEX for zpotrs.

Arrays: *a*(*lda*,*), *b*(*ldb*,*).

The array *a* contains the factor *U* or *L* (see *uplo*) as returned by potrf. The second dimension of *a* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

lda INTEGER. The leading dimension of a. *lda*≥ max(1, *n*).

ldb INTEGER. The leading dimension of b. *ldb*≥ max(1, *n*).

Output Parameters

b Overwritten by the solution matrix *X*.

info INTEGER. If *info = 0*, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `potrs` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

If *uplo* = `'U'`, the computed solution for each right-hand side *b* is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon \ |U^H||U|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for *uplo* = `'L'`. If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon$$

where $cond(A, x) = \| \ |A^{-1}||A| \ |x| \ \|_\infty / \ \|x\|_\infty \leq \ \||A^{-1}|\|_\infty \ \||A|\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$. The approximate number of floating-point operations for one right-hand side vector *b* is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?pocon`.

To refine the solution and estimate the error, call `?porfs`.

See Also

Matrix Storage Schemes

?pftrs
Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix using the Rectangular Full Packed (RFP) format.

Syntax

```
call spftrs( transr, uplo, n, nrhs, a, b, ldb, info )
call dpftrs( transr, uplo, n, nrhs, a, b, ldb, info )
call cpftrs( transr, uplo, n, nrhs, a, b, ldb, info )
call zpftrs( transr, uplo, n, nrhs, a, b, ldb, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves a system of linear equations *A*X = B* with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix *A* using the Cholesky factorization of *A*:

*A = U^T*U* for real data, *A = U^H*U* for complex data if *uplo*='U'

*A = L*L^T* for real data, *A = L*L^H* for complex data if *uplo*='L'

Before calling ?pftrs, you must call ?pftrf to compute the Cholesky factorization of *A*. *L* stands for a lower triangular matrix and *U* for an upper triangular matrix.

The matrix *A* is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

Input Parameters

transr

CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data).

If *transr* = 'N', the untransposed factor of *A*is stored in RFP format.

If *transr* = 'T', the transposed factor of *A*is stored in RFP format.

If *transr* = 'C', the conjugate-transposed factor of *A*is stored in RFP format.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', *U* is stored, where $A = U^{T}*U$ for real data, $A = U^{H}*U$ for complex data.

If *uplo* = 'L', *L* is stored, where $A = L*L^{T}$ for real data, $A = L*L^{H}$ for complex data

n

INTEGER. The order of the matrix *A*; *n*≥ 0.

nrhs

INTEGER. The number of right-hand sides, that is, the number of columns of the matrix *B*; *nrhs*≥ 0.

a, *b*

REAL for spftrs

DOUBLE PRECISION for dpftrs

COMPLEX for cpftrs

DOUBLE COMPLEX for zpftrs.

Arrays: *a*(*n**(*n* + 1)/2), *b*(*ldb*,*nrhs*).

The array *a* contains, in the RFP format, the factor *U* or *L* obtained by factorization of matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldb

INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

Output Parameters

b	The solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

See Also

Matrix Storage Schemes

?pptrs
Solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

Syntax

```
call spptrs( uplo, n, nrhs, ap, b, ldb, info )
call dpptrs( uplo, n, nrhs, ap, b, ldb, info )
call cpptrs( uplo, n, nrhs, ap, b, ldb, info )
call zpptrs( uplo, n, nrhs, ap, b, ldb, info )
call pptrs( ap, b [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations *A***X* = *B* with a packed symmetric positive-definite or, for complex data, Hermitian positive-definite matrix *A*, given the Cholesky factorization of *A*:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data	if *uplo*='U'
$A = L*L^T$ for real data, $A = L*L^H$ for complex data	if *uplo*='L'

where *L* is a lower triangular matrix and *U* is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*.

Before calling this routine, you must call ?pptrf to compute the Cholesky factorization of *A*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', *U* is stored, where $A = U^T*U$ for real data, $A = U^H*U$ for complex data. If *uplo* = 'L', *L* is stored, where $A = L*L^T$ for real data, $A = L*L^H$ for complex data
n	INTEGER. The order of matrix *A*; *n*≥ 0.
nrhs	INTEGER. The number of right-hand sides (*nrhs*≥ 0).
ap, *b*	REAL for spptrs

DOUBLE PRECISION for dpptrs

COMPLEX for cpptrs

DOUBLE COMPLEX for zpptrs.

Arrays: *ap*(*), *b*(*ldb*,*)

The size of *ap* must be at least max(1,*n*(*n*+1)/2).

The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).

ldb — INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b — Overwritten by the solution matrix *X*.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pptrs interface are as follows:

ap — Holds the array *A* of size (*n**(*n*+1)/2).

b — Holds the matrix *B* of size (*n*, *nrhs*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', the computed solution for each right-hand side *b* is the exact solution of a perturbed system of equations (*A* + *E*)*x* = *b*, where

$$|E| \leq c(n)\varepsilon\ |U^H||U|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon$$

where cond(*A*, *x*) = || $|A^{-1}||A|$ $|x|$ $||_\infty$ / $||x||_\infty \leq ||A^{-1}||_\infty ||A||_\infty = \kappa_\infty(A)$.

Note that cond(*A*, *x*) can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?ppcon.

To refine the solution and estimate the error, call ?pprfs.

See Also

Matrix Storage Schemes

?pbtrs
Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite band coefficient matrix.

Syntax

```
call spbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call pbtrs( ab, b [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for real data a system of linear equations *A*X = B* with a symmetric positive-definite or, for complex data, Hermitian positive-definite *band* matrix *A*, given the Cholesky factorization of *A*:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data	if *uplo*='U'
$A = L*L^T$ for real data, $A = L*L^H$ for complex data	if *uplo*='L'

where *L* is a lower triangular matrix and *U* is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*.

Before calling this routine, you must call ?pbtrf to compute the Cholesky factorization of *A* in the band storage form.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', *U* is stored in *ab*, where $A = U^T*U$ for real matrices and $A = U^H*U$ for complex matrices.

If *uplo* = 'L', *L* is stored in *ab*, where $A = L*L^T$ for real matrices and $A = L*L^H$ for complex matrices.

n	INTEGER. The order of matrix *A*; *n*≥ 0.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; *kd*≥ 0.
nrhs	INTEGER. The number of right-hand sides; *nrhs*≥ 0.
ab, *b*	REAL for spbtrs DOUBLE PRECISION for dpbtrs COMPLEX for cpbtrs DOUBLE COMPLEX for zpbtrs. Arrays: *ab*(*ldab*,*), *b*(*ldb*,*). The array *ab* contains the Cholesky factor, as returned by the factorization routine, in *band storage* form. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *ab* must be at least max(1, *n*), and the second dimension of *b* at least max(1,*nrhs*).
ldab	INTEGER. The leading dimension of the array *ab*; *ldab*≥*kd* +1.
ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pbtrs interface are as follows:

ab	Holds the array *A* of size (*kd*+1,*n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kd + 1)\varepsilon P|U^H||U| \text{ or } |E| \leq c(kd + 1)\varepsilon P|L^H||L|$$

$c(k)$ is a modest linear function of k, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kd + 1)\, cond\,(A, x)\varepsilon$$

where cond(*A*, *x*) = || $|A^{-1}||A|\,|x|$ $||_\infty$ / $||x||_\infty \leq ||A^{-1}||_\infty\, ||A||_\infty = \kappa_\infty(A)$.

Note that cond(*A*, *x*) can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is 4*n***kd* for real flavors and 16*n***kd* for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?pbcon.

To refine the solution and estimate the error, call ?pbrfs.

See Also

Matrix Storage Schemes

?pttrs
Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal coefficient matrix using the factorization computed by ?pttrf.

Syntax

```
call spttrs( n, nrhs, d, e, b, ldb, info )
call dpttrs( n, nrhs, d, e, b, ldb, info )
call cpttrs( uplo, n, nrhs, d, e, b, ldb, info )
call zpttrs( uplo, n, nrhs, d, e, b, ldb, info )
call pttrs( d, e, b [,info] )
call pttrs( d, e, b [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* a system of linear equations *A***X* = *B* with a symmetric (Hermitian) positive-definite tridiagonal matrix *A*. Before calling this routine, call ?pttrf to compute the $L*D*L^T$ or U^T*D*U for real data and the $L*D*L^H$ or U^H*D*U factorization of *A* for complex data.

Input Parameters

uplo

CHARACTER*1. Used for cpttrs/zpttrs only. Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix *A* is stored and how *A* is factored:

If *uplo* = 'U', the array *e* stores the conjugated values of the superdiagonal of *U*, and *A* is factored as U^H*D*U.

If *uplo* = 'L', the array *e* stores the subdiagonal of *L*, and *A* is factored as $L*D*L^H$.

n — INTEGER. The order of *A*; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides, that is, the number of columns of the matrix *B*; $nrhs \geq 0$.

d — REAL for spttrs, cpttrs

DOUBLE PRECISION for dpttrs, zpttrs.

Array, dimension (*n*). Contains the diagonal elements of the diagonal matrix *D* from the factorization computed by ?pttrf.

e, *b* — REAL for spttrs

DOUBLE PRECISION for dpttrs

COMPLEX for cpttrs

DOUBLE COMPLEX for zpttrs.

Arrays: *e*(*n* -1), *b*(*ldb*, *nrhs*).

The array *e* contains the (*n* - 1) sub-diagonal elements of the unit bidiagonal factor *L* or the conjugated values of the superdiagonal of *U* from the factorization computed by ?pttrf (see *uplo*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldb — INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b — Overwritten by the solution matrix *X*.

info — INTEGER. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pttrs interface are as follows:

d — Holds the vector of length *n*.

e — Holds the vector of length (*n*-1).

b — Holds the matrix *B* of size (*n*, *nrhs*).

uplo — Used in complex flavors only. Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?sytrs
Solves a system of linear equations with a UDU^T- or LDL^T-factored symmetric coefficient matrix.

Syntax

```
call ssytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call dsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call csytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call sytrs( a, b, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations *A*X = B* with a symmetric matrix *A*, given the Bunch-Kaufman factorization of *A*:

if *uplo*='U', $A = U*D*U^T$

if *uplo*='L', $A = L*D*L^T$,

where *U* and *L* are upper and lower triangular matrices with unit diagonal and *D* is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*. You must supply to this routine the factor *U* (or *L*) and the array *ipiv* returned by the factorization routine ?sytrf.

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization *A* = *U*D*U*^{T}.
If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization *A* = *L*D*L*^{T}.

n
INTEGER. The order of matrix *A*; *n*≥ 0.

nrhs
INTEGER. The number of right-hand sides; *nrhs*≥ 0.

ipiv
INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?sytrf.

a, *b*
REAL for ssytrs
DOUBLE PRECISION for dsytrs
COMPLEX for csytrs
DOUBLE COMPLEX for zsytrs.
Arrays: *a*(*lda*,*), *b*(*ldb*,*).

	The array *a* contains the factor *U* or *L* (see *uplo*). The second dimension of *a* must be at least `max(1,n)`. The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *b* must be at least `max(1,nrhs)`.
lda	`INTEGER`. The leading dimension of *a*; *lda*≥ `max(1, n)`.
ldb	`INTEGER`. The leading dimension of *b*; *ldb*≥ `max(1, n)`.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sytrs` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \le c(n)\varepsilon P|U||D||U^T|P^T \text{ or } |E| \le c(n)\varepsilon P|L||D||U^T|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\, cond\,(A, x)\varepsilon$$

where $cond(A, x) = \| |A^{-1}||A|\,|x|\,\|_\infty / \|x\|_\infty \le \||A^{-1}|\|_\infty \||A|\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?sycon.

To refine the solution and estimate the error, call ?syrfs.

See Also

Matrix Storage Schemes

?sytrs_aa

*Solves a system of linear equations A * X = B with a symmetric matrix.*

```
call ssytrs_aa(uplo, n, nrhs, A, lda, ipiv, B, ldb, work, lwork, info)
call dsytrs_aa(uplo, n, nrhs, A, lda, ipiv, B, ldb, work, lwork, info)
call csytrs_aa(uplo, n, nrhs, A, lda, ipiv, B, ldb, work, lwork, info)
call zsytrs_aa(uplo, n, nrhs, A, lda, ipiv, B, ldb, work, lwork, info)
```

Description

?sytrs_aa solves a system of linear equations A * X = B with a symmetric matrix A using the factorization A = U*T*U^T or A = L*T*L^T computed by ?sytrf_aa.

Input Parameters

uplo

CHARACTER*1

Specifies whether the details of the factorization are stored as an upper or lower triangular matrix.

- = 'U': Upper triangular; the form is A = U*T*U^T.
- = 'L': Lower triangular; the form is A = L*T*L^T.

n

INTEGER

The order of the matrix A. $n \geq 0$.

nrhs

INTEGER

The number of right-hand sides; that is, the number of columns of the matrix B. $nrhs \geq 0$.

A

REAL for ssytrs_aa

DOUBLE COMPLEX for dsytrs_aa

COMPLEX for csytrs_aa

COMPLEX*16 for zsytrs_aa

Array, dimension (*lda*,*n*). Details of factors computed by ?sytrf_aa.

lda

INTEGER

The leading dimension of the array *A*.$lda \geq \max(1, n)$.

ipiv

INTEGER

Array, dimension (*n*). Details of the interchanges as computed by ?sytrf_aa.

B

REAL for ssytrs_aa

DOUBLE COMPLEX for dsytrs_aa

COMPLEX for csytrs_aa

COMPLEX*16 for zsytrs_aa

Array, dimension (*ldb*,*nrhs*). On entry, the right-hand side matrix B.

ldb INTEGER

The leading dimension of the array *B*. *ldb* ≥ max(1, *n*).

work Array, dimension (MAX(1,*lwork*)).

REAL for ssytrs_aa

DOUBLE COMPLEX for dsytrs_aa

COMPLEX for csytrs_aa

COMPLEX*16 for zsytrs_aa

lwork INTEGER

The length of the array *work*.

Output Parameters

B REAL for ssytrs_aa

DOUBLE COMPLEX for dsytrs_aa

COMPLEX for csytrs_aa

COMPLEX*16 for zsytrs_aa

On exit, the solution matrix X.

info INTEGER

- If *info* = 0: Successful exit.
- < 0: If *info* = –*i*, the *i*th argument had an illegal value.

?sytrs_rook

Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix.

Syntax

```
call ssytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call dsytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call csytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zsytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call sytrs_rook( a, b, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves a system of linear equations $A*X = B$ with a symmetric matrix A, using the factorization $A = U*D*U^T$ or $A = L*D*L^T$ computed by ?sytrf_rook.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the factorization is of the form $A = U*D*U^T$. If *uplo* = 'L', the factorization is of the form $A = L*D*L^T$.
n	INTEGER. The order of matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?sytrf_rook.
a, *b*	REAL for ssytrs_rook DOUBLE PRECISION for dsytrs_rook COMPLEX for csytrs_rook DOUBLE COMPLEX for zsytrs_rook. Arrays: *a*(*lda*,*n*), *b*(*ldb*,*nrhs*). The array *a* contains the block diagonal matrix *D* and the multipliers used to obtain *U* or *L* as computed by ?sytrf_rook (see *uplo*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations.
lda	INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.
ldb	INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sytrs_rook interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?hetrs
Solves a system of linear equations with a UDU^T- or LDL^T-factored Hermitian coefficient matrix.

Syntax

```
call chetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zhetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call hetrs( a, b, ipiv [, uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations *A*X = B* with a Hermitian matrix *A*, given the Bunch-Kaufman factorization of *A*:

if *uplo*='U', $A = U*D*U^H$

if *uplo*='L', $A = L*D*L^H$,

where *U* and *L* are upper and lower triangular matrices with unit diagonal and *D* is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*. You must supply to this routine the factor *U* (or *L*) and the array *ipiv* returned by the factorization routine ?hetrf.

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization *A* = $U*D*U^H$.
If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization *A* = $L*D*L^H$.

n
INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs
INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ipiv
INTEGER.
Array, size at least max(1, *n*).
The *ipiv* array, as returned by ?hetrf.

a, *b*	`COMPLEX` for `chetrs` `DOUBLE COMPLEX` for `zhetrs`. Arrays: `a(lda,*)`, `b(ldb,*)`. The array *a* contains the factor *U* or *L* (see *uplo*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *a* must be at least `max(1,n)`, the second dimension of *b* at least `max(1,nrhs)`.
lda	`INTEGER`. The leading dimension of *a*; *lda*≥ `max(1, n)`.
ldb	`INTEGER`. The leading dimension of *b*; *ldb*≥ `max(1, n)`.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hetrs` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^H|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon$$

where cond(*A*, *x*) = || $|A^{-1}||A|$ $|x|$ $||_\infty$ / $||x||_\infty \leq ||A^{-1}||_\infty ||A||_\infty = \kappa_\infty(A)$.

Note that cond(*A*, *x*) can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$.

To estimate the condition number $\kappa_\infty(A)$, call ?hecon.

To refine the solution and estimate the error, call ?herfs.

See Also

Matrix Storage Schemes

?hetrs_aa
*BSolves a system of linear equations A*X = with a complex Hermitian matrix.*

```
call chetrs_aa(uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call zhetrs_aa(uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
```

Description

?hetrs_aa solves a system of linear equations *A*X* = *X* with a complex Hermitian matrix *A* using the factorization *A* = *U* * *T* * U^H or *A* = *L* * *T* * L^H computed by ?hetrf_aa.

Input Parameters

uplo	CHARACTER*1. Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. If *uplo* = 'U': Upper triangular of the form *A* = *U* * *T* * U^H. If *uplo*= 'L': Lower triangular of the form *A* = *L* * *T* * L^H.
n	INTEGER. The order of the matrix *A*. *n*≥ 0.
nrhs	INTEGER. The number of right hand sides: the number of columns of the matrix *b*. *nrhs*≥ 0.
a	COMPLEX for chetrs_aa COMPLEX*16 for zhetrs_aa Array of size (*lda*, *n*). Details of factors computed by ?hetrf_aa.
lda	INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*n*).
ipiv	INTEGER . Array of size (*n*). Details of the interchanges as computed by ?hetrf_aa.
b	COMPLEX for chetrs_aa COMPLEX*16 for zhetrs_aa Array of size (*ldb*, *nrhs*). On entry, the right hand side matrix *B*.
ldb	INTEGER. The leading dimension of the array *b*. *ldb*≥ max(1, *n*).
work	DOUBLE . Array of size (max(1, *lwork*)).
lwork	INTEGER. *lwork*≥ max(1, 3**n*-2).

Output Parameters

b	On exit, the solution matrix *X*.
info	INTEGER. If *info* = 0: successful exit. If *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?hetrs_rook

Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix.

Syntax

```
call chetrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zhetrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call hetrs_rook( a, b, ipiv [, uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for a system of linear equations $A*X = B$ with a complex Hermitian matrix *A* using the factorization $A = U*D*U^H$ or $A = L*D*L^H$ computed by ?hetrf_rook.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the factorization is of the form $A = U*D*U^H$. If *uplo* = 'L', the factorization is of the form $A = L*D*L^H$.
n	INTEGER. The order of matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?hetrf_rook.
a, *b*	COMPLEX for chetrs_rook DOUBLE COMPLEX for zhetrs_rook. Arrays: *a*(*lda*,*n*), *b*(*ldb*,*nrhs*). The array *a* contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* as computed by ?hetrf_rook (see *uplo*).

	The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations.
lda	INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hetrs_rook interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

?sytrs2
Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix.

Syntax

```
call ssytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call dsytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call csytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call zsytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call sytrs2( a,b,ipiv[,uplo][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves a system of linear equations *A*X = B* with a symmetric matrix *A* using the factorization of *A*:

if *uplo*='U', $A = U*D*U^T$

if *uplo*='L', $A = L*D*L^T$

where

- *U* and *L* are upper and lower triangular matrices with unit diagonal
- *D* is a symmetric block-diagonal matrix.

The factorization is computed by `?sytrf`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates how the input matrix *A* has been factored: If *uplo* = `'U'`, the array *a* stores the upper triangular factor *U* of the factorization *A* = $U*D*U^T$. If *uplo* = `'L'`, the array *a* stores the lower triangular factor *L* of the factorization *A* = $L*D*L^T$.
n	`INTEGER`. The order of matrix *A*; $n \geq 0$.
nrhs	`INTEGER`. The number of right-hand sides; $nrhs \geq 0$.

a, *b*	`REAL` for `ssytrs2`
	`DOUBLE PRECISION` for `dsytrs2`
	`COMPLEX` for `csytrs2`
	`DOUBLE COMPLEX` for `zsytrs2`
	Arrays: *a*(*lda*,*), *b*(*ldb*,*).
	The array *a* contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* as computed by `?sytrf`.
	The array *b* contains the right-hand side matrix *B*.
	The second dimension of *a* must be at least max(1,*n*), and the second dimension of *b* at least max(1,*nrhs*).

lda	`INTEGER`. The leading dimension of *a*; $lda \geq \max(1, n)$.
ldb	`INTEGER`. The leading dimension of *b*; $ldb \geq \max(1, n)$.
ipiv	`INTEGER`. Array of size *n*. The *ipiv* array contains details of the interchanges and the block structure of *D* as determined by `?sytrf`.

work	`REAL` for `ssytrs2`
	`DOUBLE PRECISION` for `dsytrs2`
	`COMPLEX` for `csytrs2`
	`DOUBLE COMPLEX` for `zsytrs2`
	Workspace array, size *n*.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sytrs2` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length *n*.
uplo	Indicates how the input matrix *A* has been factored. Must be `'U'` or `'L'`.

See Also

?sytrf

Matrix Storage Schemes

?hetrs2
Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix.

Syntax

```
call chetrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call zhetrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call hetrs2( a, b, ipiv [,uplo] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine solves a system of linear equations $A*X = B$ with a complex Hermitian matrix *A* using the factorization of *A*:

if *uplo*='U',	$A = U*D*U^H$
if *uplo*='L',	$A = L*D*L^H$

where

- *U* and *L* are upper and lower triangular matrices with unit diagonal
- *D* is a Hermitian block-diagonal matrix.

The factorization is computed by `?hetrf`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates how the input matrix *A* has been factored: If *uplo* = `'U'`, the array *a* stores the upper triangular factor *U* of the factorization *A* = $U*D*U^H$.

	If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization *A* = *L*D*L*^{H}.
n	INTEGER. The order of matrix *A*; *n*≥ 0.
nrhs	INTEGER. The number of right-hand sides; *nrhs*≥ 0.
a, *b*	COMPLEX for chetrs2 DOUBLE COMPLEX for zhetrs2 Arrays: *a*(*lda*,*), *b*(*ldb*,*). The array *a* contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* as computed by ?hetrf. The array *b* contains the right-hand side matrix *B*. The second dimension of *a* must be at least max(1,*n*), and the second dimension of *b* at least max(1,*nrhs*).
lda	INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).
ipiv	INTEGER. Array of size *n*. The *ipiv* array contains details of the interchanges and the block structure of *D* as determined by ?hetrf.
work	COMPLEX for chetrs2 DOUBLE COMPLEX for zhetrs2 Workspace array, size *n*.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hetrs2 interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

See Also

?hetrf

Matrix Storage Schemes

?sytrs_3
*Solves a system of linear equations A * X = B with a real or complex symmetric matrix.*

```
call ssytrs_3(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, info)
call dsytrs_3(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, info)
call csytrs_3(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, info)
call zsytrs_3(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, info)
```

Description

`?sytrs_3` solves a system of linear equations A * X = B with a real or complex symmetric matrix A using the factorization computed by `?sytrf_rk`: A = P*U*D*(U^T)*(P^T) or A = P*L*D*(L^T)*(P^T), where U (or L) is unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This algorithm uses Level 3 BLAS.

Input Parameters

uplo

`CHARACTER*1`

Specifies whether the details of the factorization are stored as an upper or lower triangular matrix:

- = `'U'`: Upper triangular; the form is A= P*U*D*(U^T)*(P^T).
- = `'L'`: Lower triangular; the form is A = P*L*D*(L^T)*(P^T).

n

`INTEGER`

The order of the matrix A. *n* ≥ 0.

nrhs

`INTEGER`

The number of right-hand sides; that is, the number of columns of the matrix B. *nrhs* ≥ 0.

A

`REAL` for `ssytrs_3`

`DOUBLE PRECISION` for `dsytrs_3`

`COMPLEX` for `csytrs_3`

`COMPLEX*16` for `zsytrs_3`

Array, dimension (*lda*,*n*). Diagonal of the block diagonal matrix D and factors U or L as computed by `?sytrf_rk`:

- *Only* diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, D(*k*,*k*) = A(*k*,*k*). Superdiagonal (or subdiagonal) elements of D should be provided on entry in array *e*.

  —and—
- If *uplo* = `'U'`, factor U in the superdiagonal part of A. If *uplo* = `'L'`, factor L in the subdiagonal part of A.

lda

`INTEGER`

The leading dimension of the array *A*.*lda* ≥ max(1, *n*).

e

`REAL` for `ssytrs_3`

DOUBLE PRECISION for dsytrs_3

COMPLEX for csytrs_3

COMPLEX*16 for zsytrs_3

Array, dimension (*n*). On entry, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', e(*i*) = D(*i*–1,*i*),*i*=2:N, and e(1) is not referenced. If *uplo* = 'L', e(*i*) = D(*i*+1,*i*), *i*=1:N–1, and e(*n*) is not referenced.

NOTE For 1-by-1 diagonal block D(*k*), where 1 ≤ k ≤ *n*, the element *e*(*k*) is not referenced in both the *uplo* = 'U' and *uplo* = 'L' cases.

ipiv

INTEGER

Array, dimension (*n*). Details of the interchanges and the block structure of D as determined by ?sytrf_rk.

B

REAL for ssytrs_3

DOUBLE PRECISION for dsytrs_3

COMPLEX for csytrs_3

COMPLEX*16 for zsytrs_3

On entry, the right-hand side matrix B.

The second dimension of *B* must be at least max(1, *nrhs*).

ldb

INTEGER

The leading dimension of the array *B*. *ldb* ≥ max(1, *n*).

Output Parameters

B

REAL for ssytrs_3

DOUBLE PRECISION for dsytrs_3

COMPLEX for csytrs_3

COMPLEX*16 for zsytrs_3

On exit, the solution matrix X.

info

INTEGER

- = 0: successful exit.
- < 0: If *info* = –*i*, the *i*th argument had an illegal value.

?hetrs_3

*Solves a system of linear equations A * X = B with a complex Hermitian matrix using the factorization computed by* ?hetrf_rk.

```
call chetrs_3(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, info)
call zhetrs_3(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, info)
```

Description

?hetrs_3 solves a system of linear equations A * X = B with a complex Hermitian matrix A using the factorization computed by ?hetrf_rk: A = P*U*D*(U^H)*(P^T) or A = P*L*D*(L^H)*(P^T), where U (or L) is unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This algorithm uses Level 3 BLAS.

Input Parameters

uplo CHARACTER*1

Specifies whether the details of the factorization are stored as an upper or lower triangular matrix:

- = 'U': Upper triangular; form is A = P*U*D*(U^H)*(P^T).
- = 'L': Lower triangular; form is A = P*L*D*(L^H)*(P^T).

n INTEGER

The order of the matrix A. *n* ≥ 0.

nrhs INTEGER

The number of right-hand sides; that is, the number of columns in the matrix B. *nrhs* ≥ 0.

A COMPLEX for chetrs_3

COMPLEX*16 for zhetrs_3

Array, dimension (*lda*,*n*). Diagonal of the block diagonal matrix D and factor U or L as computed by ?hetrf_rk:

- *Only* diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, D(*k*,*k*) = A(*k*,*k*). Superdiagonal (or subdiagonal) elements of D should be provided on entry in array *e*.
- If *uplo* = 'U', factor U in the superdiagonal part of A. If *uplo* = 'L', factor L in the subdiagonal part of A.

lda INTEGER

The leading dimension of the array *A*.*lda* ≥ max(1, *n*).

e COMPLEX for chetrs_3

COMPLEX*16 for zhetrs_3

Array, dimension (*n*). On entry, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', e(*i*) = D(*i*–1,*i*),*i*=2:N, and e(1) is not referenced. If *uplo* = 'L', e(*i*) = D(*i*+1,*i*),*i*=1:N–1, and e(*n*) is not referenced.

> **NOTE** For 1-by-1 diagonal block D(*k*), where 1 ≤ *k* ≤ *n*, the element *e*(*k*) is not referenced in both the *uplo* = 'U' and *uplo* = 'L' cases.

ipiv INTEGER

Array, dimension (*n*). Details of the interchanges and the block structure of D as determined by ?hetrf_rk.

B — COMPLEX for chetrs_3

COMPLEX*16 for zhetrs_3

On entry, the right-hand side matrix B.

The second dimension of *B* must be at least max(1, *nrhs*).

ldb — INTEGER

The leading dimension of the array *B*. *ldb* ≥ max(1, *n*).

Output Parameters

B — COMPLEX for chetrs_3

COMPLEX*16 for zhetrs_3

On exit, the solution matrix X.

info — INTEGER

- = 0: Successful exit.
- < 0: If *info* = *-i*, the *i*th argument had an illegal value.

?sptrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix using packed storage.

Syntax

```
call ssptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call csptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call sptrs( ap, b, ipiv [, uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations *A*X = B* with a symmetric matrix *A*, given the Bunch-Kaufman factorization of *A*:

if *uplo*='U', $A = U*D*U^T$

if *uplo*='L', $A = L*D*L^T$,

where *U* and *L* are upper and lower *packed* triangular matrices with unit diagonal and *D* is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*. You must supply the factor *U* (or *L*) and the array *ipiv* returned by the factorization routine ?sptrf.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the array *ap* stores the packed factor *U* of the factorization $A = U*D*U^T$. If *uplo* = 'L', the array *ap* stores the packed factor *L* of the factorization $A = L*D*L^T$.
n	INTEGER. The order of matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?sptrf.
ap	REAL for ssptrs DOUBLE PRECISION for dsptrs COMPLEX for csptrs DOUBLE COMPLEX for zsptrs. The dimension of array *ap* must be at least max(1, *n*(*n*+1)/2). The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes).
b	REAL for ssptrs DOUBLE PRECISION for dsptrs COMPLEX for csptrs DOUBLE COMPLEX for zsptrs. The array *b*(*ldb*,*) contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *b* must be at least max(1, *nrhs*).
ldb	INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sptrs interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*, *nrhs*).

ipiv	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \le c(n)\varepsilon P|U||D||U^T|P^T \text{ or } |E| \le c(n)\varepsilon P|L||D||L^T|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\, cond\,(A, x)\varepsilon$$

where $cond(A,x) = \| |A^{-1}||A|\,|x|\,\|_\infty / \||x|\|_\infty \le \||A^{-1}|\|_\infty \||A|\|_\infty = \kappa_\infty(A)$.

Note that $cond(A,x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?spcon.

To refine the solution and estimate the error, call ?sprfs.

See Also

Matrix Storage Schemes

?hptrs
Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix using packed storage.

Syntax

```
call chptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call hptrs( ap, b, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations $A*X = B$ with a Hermitian matrix *A*, given the Bunch-Kaufman factorization of *A*:

if *uplo*='U', $A = U*D*U^H$

if *uplo*='L', $A = L*D*L^H$,

where *U* and *L* are upper and lower *packed* triangular matrices with unit diagonal and *D* is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*.

You must supply to this routine the arrays *ap* (containing *U* or *L*)and *ipiv* in the form returned by the factorization routine ?hptrf.

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the array *ap* stores the packed factor *U* of the factorization $A = U*D*U^H$. If *uplo* = 'L', the array *ap* stores the packed factor *L* of the factorization $A = L*D*L^H$.

n
INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs
INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ipiv
INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?hptrf.

ap
COMPLEX for chptrs
DOUBLE COMPLEX for zhptrs.
The dimension of array *ap*(*) must be at least max(1,*n*(*n*+1)/2). The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes).

b
COMPLEX for chptrs
DOUBLE COMPLEX for zhptrs.
The array *b*(*ldb*,*) contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *b* must be at least max(1, *nrhs*).

ldb
INTEGER. The leading dimension of *b*; $ldb \geq$ max(1, *n*).

Output Parameters

b
Overwritten by the solution matrix *X*.

info
INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hptrs interface are as follows:

ap	Holds the array *A* of size `(n*(n+1)/2)`.
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^H|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon$$

where $cond(A,x) = \| |A^{-1}||A|\,|x| \|_\infty / \||x|\|_\infty \leq \||A^{-1}|\|_\infty \||A|\|_\infty = \kappa_\infty(A)$.

Note that $cond(A,x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?hpcon`.

To refine the solution and estimate the error, call `?hprfs`.

See Also

Matrix Storage Schemes

?trtrs
Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.

Syntax

```
call strtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call dtrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ctrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ztrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call trtrs( a, b [,uplo] [, trans] [,diag] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine solves for *X* the following systems of linear equations with a triangular matrix *A*, with multiple right-hand sides stored in *B*:

*A*X = B*	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether *A* is upper or lower triangular: If *uplo* = 'U', then *A* is upper triangular. If *uplo* = 'L', then *A* is lower triangular.
trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. If *trans* = 'N', then *A*X = B* is solved for *X*. If *trans* = 'T', then $A^T*X = B$ is solved for *X*. If *trans* = 'C', then $A^H*X = B$ is solved for *X*.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', then *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.
n	INTEGER. The order of *A*; the number of rows in *B*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
a	REAL for strtrs DOUBLE PRECISION for dtrtrs COMPLEX for ctrtrs DOUBLE COMPLEX for ztrtrs. The array *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1,*n*).
b	REAL for strtrs DOUBLE PRECISION for dtrtrs COMPLEX for ctrtrs DOUBLE COMPLEX for ztrtrs. The array *b*(*ldb*,*) contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* at least max(1,*nrhs*).

lda	INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* =0, the execution is successful. If *info* = -i, the i-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine trtrs interface are as follows:

a	Stands for argument *ap* in FORTRAN 77 interface. Holds the matrix *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations (*A* + *E*)*x* = *b*, where

$$|E| \leq c(n)\varepsilon\ |A|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision. If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\ cond\ (A, x)\varepsilon \text{ provided } c(n)\ cond\ (A, x)\varepsilon < 1$$

where $cond(A, x) = \| |A^{-1}||A|\ |x|\ \|_\infty / \|x\|_\infty \leq \||A^{-1}|\|_\infty\ \||A|\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?trcon.

To estimate the error in the solution, call ?trrfs.

See Also

Matrix Storage Schemes

?tptrs
Solves a system of linear equations with a packed triangular coefficient matrix, with multiple right-hand sides.

Syntax

```
call stptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call dtptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call ctptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call ztptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call tptrs( ap, b [,uplo] [, trans] [,diag] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the following systems of linear equations with a packed triangular matrix *A*, with multiple right-hand sides stored in *B*:

$A*X = B$	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether *A* is upper or lower triangular: If *uplo* = 'U', then *A* is upper triangular. If *uplo* = 'L', then *A* is lower triangular.
trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. If *trans* = 'N', then *A*X = B* is solved for *X*. If *trans* = 'T', then $A^T*X = B$ is solved for *X*. If *trans* = 'C', then $A^H*X = B$ is solved for *X*.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ap*.
n	INTEGER. The order of *A*; the number of rows in *B*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ap	REAL for stptrs DOUBLE PRECISION for dtptrs

	COMPLEX for ctptrs DOUBLE COMPLEX for ztptrs. The dimension of array ap(*) must be at least max(1,n(n+1)/2). The array *ap* contains the matrix *A* in *packed storage* (see Matrix Storage Schemes).
b	REAL for stptrs DOUBLE PRECISION for dtptrs COMPLEX for ctptrs DOUBLE COMPLEX for ztptrs. The array b(ldb,*) contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *b* must be at least max(1, nrhs).
ldb	INTEGER. The leading dimension of *b*; ldb≥ max(1, n).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If info=0, the execution is successful. If info = -i, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine tptrs interface are as follows:

ap	Holds the array *A* of size (n*(n+1)/2).
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations (*A* + *E*)*x* = *b*, where

$$|E| \leq c(n)\varepsilon\ |A|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\ (A, x)\varepsilon \text{ provided } c(n)\, cond\ (A, x)\varepsilon < 1$$

where $cond(A, x) = \| |A^{-1}||A|\ |x|\ \|_\infty / \|x\|_\infty \leq \||A^{-1}|\|_\infty \||A|\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?tpcon`.

To estimate the error in the solution, call `?tprfs`.

See Also

Matrix Storage Schemes

?tbtrs
Solves a system of linear equations with a band triangular coefficient matrix, with multiple right-hand sides.

Syntax

```
call stbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call dtbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ctbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ztbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call tbtrs( ab, b [,uplo] [, trans] [,diag] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine solves for *X* the following systems of linear equations with a band triangular matrix *A*, with multiple right-hand sides stored in *B*:

$A*X = B$	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether *A* is upper or lower triangular: If *uplo* = 'U', then *A* is upper triangular. If *uplo* = 'L', then *A* is lower triangular.

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. If *trans* = 'N', then *A*X* = *B* is solved for *X*. If *trans* = 'T', then $A^T*X = B$ is solved for *X*. If *trans* = 'C', then $A^H*X = B$ is solved for *X*.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ab*.
n	INTEGER. The order of *A*; the number of rows in *B*; $n \geq 0$.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ab	REAL for stbtrs DOUBLE PRECISION for dtbtrs COMPLEX for ctbtrs DOUBLE COMPLEX for ztbtrs. The array *ab*(*ldab*,*) contains the matrix *A* in *band storage* form. The second dimension of *ab* must be at least max(1, *n*).
b	REAL for stbtrs DOUBLE PRECISION for dtbtrs COMPLEX for ctbtrs DOUBLE COMPLEX for ztbtrs. The array *b*(*ldb*,*) contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* at least max(1,*nrhs*).
ldab	INTEGER. The leading dimension of *ab*; $ldab \geq kd + 1$.
ldb	INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `tbtrs` interface are as follows:

ab	Holds the array *A* of size `(kd+1,n)`
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.
diag	Must be `'N'` or `'U'`. The default value is `'N'`.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon|A|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision. If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon \text{ provided } c(n)\, cond\,(A, x)\varepsilon < 1$$

where $cond(A, x) = \| |A^{-1}||A|\,|x| \|_\infty / \|x\|_\infty \leq \||A^{-1}|\|_\infty \||A|\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is $2n*kd$ for real flavors and $8n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?tbcon`.

To estimate the error in the solution, call `?tbrfs`.

See Also

Matrix Storage Schemes

Estimating the Condition Number: LAPACK Computational Routines

This section describes the LAPACK routines for estimating the *condition number* of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations (see Error Analysis). Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

?gecon
Estimates the reciprocal of the condition number of a general matrix in the 1-norm or the infinity-norm.

Syntax

```
call sgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call dgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call cgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
call zgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
```

```
call gecon( a, anorm, rcond [,norm] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a general matrix *A* in the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$

$\kappa_\infty(A) = ||A||_\infty||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H)$.

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as *rcond* = 1 / (||*A*|| ||A^{-1}||).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?getrf to compute the *LU* factorization of *A*.

Input Parameters

norm	CHARACTER*1. Must be '1' or 'O' or 'I'. If *norm* = '1' or 'O', then the routine estimates the condition number of matrix *A* in 1-norm. If *norm* = 'I', then the routine estimates the condition number of matrix *A* in infinity-norm.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
a, *work*	REAL for sgecon DOUBLE PRECISION for dgecon COMPLEX for cgecon DOUBLE COMPLEX for zgecon. Arrays: *a*(*lda*,*), *work*(*). The array *a* contains the *LU*-factored matrix *A*, as returned by ?getrf. The second dimension of *a* must be at least max(1,*n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 4*n) for real flavors and max(1, 2*n) for complex flavors.
anorm	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the *original* matrix *A* (see Description).
lda	INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, size at least max(1, *n*).
rwork	REAL for cgecon DOUBLE PRECISION for zgecon. Workspace array, size at least max(1, 2*n).

Output Parameters

rcond	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gecon` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
norm	Must be `'1'`, `'O'`, or `'I'`. The default value is `'1'`.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A*x = b* or A^Hx = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately 2*n^2 floating-point operations for real flavors and 8*n^2 for complex flavors.

See Also

Matrix Storage Schemes

?gbcon
Estimates the reciprocal of the condition number of a band matrix in the 1-norm or the infinity-norm.

Syntax

```
call sgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info )
call dgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info )
call cgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info )
call zgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info )
call gbcon( ab, ipiv, anorm, rcond [,kl] [,norm] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a general band matrix *A* in the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$

$\kappa_\infty(A) = ||A||_\infty||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H)$.

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as `rcond = 1 / (||A|| ||A^-1||)`.

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call `?gbtrf` to compute the *LU* factorization of *A*.

Input Parameters

norm	`CHARACTER*1`. Must be `'1'` or `'O'` or `'I'`. If *norm* = `'1'` or `'O'`, then the routine estimates the condition number of matrix *A* in 1-norm. If *norm* = `'I'`, then the routine estimates the condition number of matrix *A* in infinity-norm.
n	`INTEGER`. The order of the matrix *A*; $n \geq 0$.
kl	`INTEGER`. The number of subdiagonals within the band of *A*; $kl \geq 0$.
ku	`INTEGER`. The number of superdiagonals within the band of *A*; $ku \geq 0$.
ldab	`INTEGER`. The leading dimension of the array *ab*. ($ldab \geq 2*kl + ku + 1$).
ipiv	`INTEGER`. Array, size at least `max(1, n)`. The *ipiv* array, as returned by `?gbtrf`.

ab, *work*	`REAL` for `sgbcon`
	`DOUBLE PRECISION` for `dgbcon`
	`COMPLEX` for `cgbcon`
	`DOUBLE COMPLEX` for `zgbcon`.
	Arrays: `ab(ldab,*)`, `work(*)`.
	The array *ab* contains the factored band matrix *A*, as returned by `?gbtrf`.
	The second dimension of *ab* must be at least `max(1,n)`. The array *work* is a workspace for the routine.
	The dimension of *work* must be at least `max(1, 3*n)` for real flavors and `max(1, 2*n)` for complex flavors.

anorm	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. The norm of the *original* matrix *A*(see Description).
iwork	`INTEGER`. Workspace array, size at least `max(1, n)`.
rwork	`REAL` for `cgbcon` `DOUBLE PRECISION` for `zgbcon`.

Workspace array, size at least `max(1, 2*n)`.

Output Parameters

rcond

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

`INTEGER`. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gbcon` interface are as follows:

ab

Holds the array *A* of size (`2*kl+ku+1,n`).

ipiv

Holds the vector of length *n*.

norm

Must be `'1'`, `'O'`, or `'I'`. The default value is `'1'`.

kl

If omitted, assumed *kl* = *ku*.

ku

Restored as *ku* = *lda*-2*kl*-1.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than `10r`. A call to this routine involves solving a number of systems of linear equations *A*x* = *b* or A^H*x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately `2n(ku + 2kl)` floating-point operations for real flavors and `8n(ku + 2kl)` for complex flavors.

See Also

Matrix Storage Schemes

?gtcon
Estimates the reciprocal of the condition number of a tridiagonal matrix.

Syntax

```
call sgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call dgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call cgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
call zgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
call gtcon( dl, d, du, du2, ipiv, anorm, rcond [,norm] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a real or complex tridiagonal matrix *A* in the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1||A^{-1}||_1$

$\kappa_\infty(A) = ||A||_\infty||A^{-1}||_\infty$

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as *rcond* = 1 / (||*A*|| ||A^{-1}||).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?gttrf to compute the *LU* factorization of *A*.

Input Parameters

norm

CHARACTER*1. Must be '1' or 'O' or 'I'.

If *norm* = '1' or 'O', then the routine estimates the condition number of matrix *A* in 1-norm.

If *norm* = 'I', then the routine estimates the condition number of matrix *A* in infinity-norm.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

dl,*d*,*du*,*du2*

REAL for sgtcon

DOUBLE PRECISION for dgtcon

COMPLEX for cgtcon

DOUBLE COMPLEX for zgtcon.

Arrays: *dl*(*n* -1), *d*(*n*), *du*(*n* -1), *du2*(*n* -2).

The array *dl* contains the (*n* - 1) multipliers that define the matrix *L* from the *LU* factorization of *A* as computed by ?gttrf.

The array *d* contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

The array *du* contains the (*n* - 1) elements of the first superdiagonal of *U*.

The array *du2* contains the (*n* - 2) elements of the second superdiagonal of *U*.

ipiv

INTEGER.

Array, size (*n*). The array of pivot indices, as returned by ?gttrf.

anorm

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix *A*(see *Description*).

work	REAL for sgtcon DOUBLE PRECISION for dgtcon COMPLEX for cgtcon DOUBLE COMPLEX for zgtcon. Workspace array, size (2*n).
iwork	INTEGER. Workspace array, size (*n*). Used for real flavors only.

Output Parameters

rcond	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond*=0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gtcon interface are as follows:

dl	Holds the vector of length (*n*-1).
d	Holds the vector of length *n*.
du	Holds the vector of length (*n*-1).
du2	Holds the vector of length (*n*-2).
ipiv	Holds the vector of length *n*.
norm	Must be '1', 'O', or 'I'. The default value is '1'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A*x = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pocon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix.

Syntax

```
call spocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call dpocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call cpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
call zpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
call pocon( a, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix *A*:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as *rcond* = 1 / (||*A*|| ||A^{-1}||).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?potrf to compute the Cholesky factorization of *A*.

Input Parameters

n	INTEGER. The order of the matrix *A*; $n \geq 0$.
a, *work*	REAL for spocon DOUBLE PRECISION for dpocon COMPLEX for cpocon DOUBLE COMPLEX for zpocon. Arrays: *a*(*lda*,*), *work*(*). The array *a* contains the factored matrix *A*, as returned by ?potrf. The second dimension of *a* must be at least max(1,*n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.
lda	INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).
anorm	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. The norm of the *original* matrix *A* (see *Description*).
iwork	INTEGER. Workspace array, size at least max(1, *n*).
rwork	REAL for cpocon

`DOUBLE PRECISION` for `zpocon`.

Workspace array, size at least `max(1, n)`.

Output Parameters

rcond

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

`INTEGER`. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `pocon` interface are as follows:

a

Holds the matrix *A* of size (*n*, *n*).

uplo

Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?ppcon
Estimates the reciprocal of a packed symmetric (Hermitian) positive-definite matrix.

Syntax

```
call sppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call dppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call cppcon( uplo, n, ap, anorm, rcond, work, rwork, info )
call zppcon( uplo, n, ap, anorm, rcond, work, rwork, info )
call ppcon( ap, anorm, rcond [,uplo] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix *A*:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as *rcond* = 1 / (||*A*|| $||A^{-1}||$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?pptrf to compute the Cholesky factorization of *A*.

Input Parameters

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

ap, *work*

REAL for sppcon

DOUBLE PRECISION for dppcon

COMPLEX for cppcon

DOUBLE COMPLEX for zppcon.

Arrays: *ap*(*), *work*(*).

The array *ap* contains the packed factored matrix *A*, as returned by ?pptrf. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2).

The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

anorm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix *A* (see *Description*).

iwork

INTEGER. Workspace array, size at least max(1, *n*).

rwork

REAL for cppcon

DOUBLE PRECISION for zppcon.

Workspace array, size at least max(1, *n*).

Output Parameters

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ppcon interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?pbcon
Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix.

Syntax

```
call spbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call dpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call cpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
call zpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
call pbcon( ab, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix *A*:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as *rcond* = 1 / (||*A*|| $||A^{-1}||$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?pbtrf to compute the Cholesky factorization of *A*.

Input Parameters

n	INTEGER. The order of the matrix *A*; $n \geq 0$.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.
ldab	INTEGER. The leading dimension of the array *ab*. ($ldab \geq kd + 1$).
ab, *work*	REAL for spbcon DOUBLE PRECISION for dpbcon COMPLEX for cpbcon DOUBLE COMPLEX for zpbcon. Arrays: *ab*(*ldab*,*), *work*(*). The array *ab* contains the factored matrix *A* in band form, as returned by ?pbtrf. The second dimension of *ab* must be at least max(1, *n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
anorm	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the *original* matrix *A* (see *Description*).
iwork	INTEGER. Workspace array, size at least max(1, *n*).
rwork	REAL for cpbcon DOUBLE PRECISION for zpbcon. Workspace array, size at least max(1, *n*).

Output Parameters

rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pbcon interface are as follows:

ab	Holds the array *A* of size (*kd*+1,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately 4**n*(*kd* + 1) floating-point operations for real flavors and 16**n*(*kd* + 1) for complex flavors.

See Also

Matrix Storage Schemes

?ptcon
Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

```
call sptcon( n, d, e, anorm, rcond, work, info )
call dptcon( n, d, e, anorm, rcond, work, info )
call cptcon( n, d, e, anorm, rcond, work, info )
call zptcon( n, d, e, anorm, rcond, work, info )
call ptcon( d, e, anorm, rcond [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the reciprocal of the condition number (in the 1-norm) of a real symmetric or complex Hermitian positive-definite tridiagonal matrix using the factorization $A = L*D*L^T$ for real flavors and $A = L*D*L^H$ for complex flavors or $A = U^T*D*U$ for real flavors and $A = U^H*D*U$ for complex flavors computed by ?pttrf :

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

The norm $||A^{-1}||$ is computed by a direct method, and the reciprocal of the condition number is computed as *rcond* = 1 / (||*A*|| $||A^{-1}||$).

Before calling this routine:

- compute *anorm* as $||A||_1 = \max_j \Sigma_i |a_{ij}|$
- call ?pttrf to compute the factorization of *A*.

Input Parameters

n	INTEGER. The order of the matrix *A*; $n \geq 0$.
d, *work*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Arrays, dimension (*n*).

The array *d* contains the *n* diagonal elements of the diagonal matrix *D* from the factorization of *A*, as computed by ?pttrf ;

work is a workspace array.

e

REAL for sptcon

DOUBLE PRECISION for dptcon

COMPLEX for cptcon

DOUBLE COMPLEX for zptcon.

Array, size (*n* -1).

Contains off-diagonal elements of the unit bidiagonal factor *U* or *L* from the factorization computed by ?pttrf .

anorm

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The 1- norm of the *original* matrix *A* (see *Description*).

Output Parameters

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gtcon interface are as follows:

d

Holds the vector of length *n*.

e

Holds the vector of length (*n*-1).

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately 4**n*(*kd* + 1) floating-point operations for real flavors and 16**n*(*kd* + 1) for complex flavors.

?sycon
Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

```
call ssycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call dsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call csycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call sycon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a symmetric matrix *A*:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is symmetric, $\kappa_\infty(A) = \kappa_1(A)$).

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as *rcond* = 1 / (||*A*|| $||A^{-1}||$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?sytrf to compute the factorization of *A*.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization *A* = $U*D*U^T$.

If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization *A* = $L*D*L^T$.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

a, *work*

REAL for ssycon

DOUBLE PRECISION for dsycon

COMPLEX for csycon

DOUBLE COMPLEX for zsycon.

Arrays: *a*(*lda*,*), *work*(*).

The array *a* contains the factored matrix *A*, as returned by ?sytrf. The second dimension of *a* must be at least max(1,*n*).

The array *work* is a workspace for the routine.

	The dimension of *work* must be at least `max(1, 2*n)`.
lda	`INTEGER`. The leading dimension of *a*; *lda*≥ `max(1, n)`.
ipiv	`INTEGER`. Array, size at least `max(1, n)`. The array *ipiv*, as returned by `?sytrf`.
anorm	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. The norm of the *original* matrix *A* (see *Description*).
iwork	`INTEGER`. Workspace array, size at least `max(1, n)`.

Output Parameters

rcond	`REAL` for single precision flavors `DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sycon` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?sycon_rook
Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

```
call ssycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call dsycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call csycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zsycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call sycon_rook( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a symmetric matrix *A*:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is symmetric, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?sytrf_rook to compute the factorization of *A*.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization $A = U*D*U^T$.

If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization $A = L*D*L^T$.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

a, *work*

REAL for ssycon_rook

DOUBLE PRECISION for dsycon_rook

COMPLEX for csycon_rook

DOUBLE COMPLEX for zsycon_rook.

Arrays: *a*(*lda*,*), *work*(*).

The array *a* contains the factored matrix *A*, as returned by ?sytrf_rook. The second dimension of *a* must be at least max(1,*n*).

The array *work* is a workspace for the routine.

The dimension of *work* must be at least max(1, 2*n).

lda

INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ipiv

INTEGER. Array, size at least max(1, *n*).

The array *ipiv*, as returned by ?sytrf_rook.

anorm	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. The norm of the *original* matrix *A* (see *Description*).
iwork	`INTEGER`. Workspace array, size at least `max(1, n)`.

Output Parameters

rcond	`REAL` for single precision flavors `DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sycon_rook` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A*x = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?sycon_3

Estimates the reciprocal of the condition number (in the 1-norm) of a real or complex symmetric matrix A using the factorization computed by `?sytrf_rk`.

```
call ssycon_3(uplo, n, A, lda, e, ipiv, anorm, rcond, work, iwork, info)
call dsycon_3(uplo, n, A, lda, e, ipiv, anorm, rcond, work, iwork, info)
call csycon_3(uplo, n, A, lda, e, ipiv, anorm, rcond, work, info)
call zsycon_3(uplo, n, A, lda, e, ipiv, anorm, rcond, work, info)
```

Description

`?sycon_3` estimates the reciprocal of the condition number (in the 1-norm) of a real or complex symmetric matrix A using the factorization computed by `?sytrf_rk`. A = P*U*D*(U^T)*(P^T) or A = P*L*D*(L^T)*(P^T), where U (or L) is unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as *rcond* = 1 / (*anorm* * norm(inv(A))).

This routine uses BLAS3 solver `?sytrs_3`.

Input Parameters

uplo

`CHARACTER*1`

Specifies whether the details of the factorization are stored as an upper or lower triangular matrix:

- = `'U'`: Upper triangular. The form is A = P*U*D*(U^T)*(P^T).
- = `'L'`: Lower triangular. The form is A = P*L*D*(L^T)*(P^T).

n

`INTEGER`

The order of the matrix A. *n* ≥ 0.

A

`REAL` for `ssycon_3`

`DOUBLE PRECISION` for `dsycon_3`

`COMPLEX` for `csycon_3`

`COMPLEX*16` for `zsycon_3`

Array, dimension (*lda*,*n*). Diagonal of the block diagonal matrix D and factors U or L as computed by `?sytrf_rk`:

- *Only* diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, D(*k*,*k*) = A(*k*,*k*). Superdiagonal (or subdiagonal) elements of D should be provided on entry in array *e*).

  —and—
- If *uplo* = `'U'`, factor U in the superdiagonal part of A. If *uplo* = `'L'`, factor L in the subdiagonal part of A.

lda

`INTEGER`

The leading dimension of the array *A*.*lda* ≥ max(1, *n*).

e

`REAL` for `ssycon_3`

`DOUBLE PRECISION` for `dsycon_3`

`COMPLEX` for `csycon_3`

`COMPLEX*16` for `zsycon_3`

Array, dimension (*n*). On entry, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = `'U'`, e(*i*) = D(*i*–1,*i*), *i*=2:N, and e(1) is not referenced. If *uplo* = `'L'`, e(*i*) = D(*i*+1,*i*), *i*=1:N–1, and e(*n*) is not referenced.

NOTE For 1-by-1 diagonal block D(k), where $1 \le k \le n$, the element $e(k)$ is not referenced in both the *uplo* = 'U' and *uplo* = 'L' cases.

ipiv INTEGER

Array, dimension (n). Details of the interchanges and the block structure of D as determined by ?sytrf_rk.

anorm REAL for ssycon_3

DOUBLE PRECISION for dsycon_3

REAL for csycon_3

DOUBLE PRECISION for zsycon_3

The 1-norm of the original matrix A.

Output Parameters

rcond REAL for ssycon_3

DOUBLE PRECISION for dsycon_3

REAL for csycon_3

DOUBLE PRECISION for zsycon_3

The reciprocal of the condition number of the matrix A, computed as *rcond* = 1/(*anorm* * *AINVNM*), where *AINVNM* is an estimate of the 1-norm of inv(A) computed in this routine.

work REAL for ssycon_3

DOUBLE PRECISION for dsycon_3

COMPLEX for csycon_3

COMPLEX*16 for zsycon_3

Array, dimension (2*n)

iwork INTEGER

Array, dimension (n).

info INTEGER

- = 0: Successful exit.
- < 0: If *info* = -*i*, the i^{th} argument had an illegal value.

?hecon

Estimates the reciprocal of a Hermitian matrix.

Syntax

```
call checon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zhecon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
```

```
call hecon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix *A*:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?hetrf to compute the factorization of *A*.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization $A = U*D*U^H$.

If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization $A = L*D*L^H$.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

a, *work*

COMPLEX for checon

DOUBLE COMPLEX for zhecon.

Arrays: *a*(*lda*,*), *work*(*).

The array *a* contains the factored matrix *A*, as returned by ?hetrf. The second dimension of *a* must be at least max(1,*n*).

The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 2*n).

lda

INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).

ipiv

INTEGER. Array, size at least max(1, *n*).

The array *ipiv*, as returned by ?hetrf.

anorm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix *A* (see *Description*).

Output Parameters

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

	An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hecon interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
ipiv	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A*x = b*; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

See Also

Matrix Storage Schemes

?hecon_rook
Estimates the reciprocal of the condition number of a Hermitian matrix using factorization obtained with one of the bounded diagonal pivoting methods (max 2 interchanges).

Syntax

```
call checon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zhecon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call hecon_rook( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix *A* using the factorization $A = U*D*U^H$ or $A = L*D*L^H$ computed by hetrf_rook.

An estimate is obtained for norm(A^{-1}), and the reciprocal of the condition number is computed as *rcond* = 1/(*anorm**norm(A^{-1})).

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization *A* = $U*D*U^H$.
If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization *A* = $L*D*L^H$.

n
INTEGER. The order of matrix *A*; $n \geq 0$.

a, *work*
COMPLEX for checon_rook
COMPLEX*16 for zhecon_rook.
Arrays: *a*(*lda*,*n*), *work*(*).
The array *a* contains the factored matrix *A*, as returned by ?hetrf_rook. The second dimension of *a* must be at least max(1,*n*).
The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 2*n).

lda
INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ipiv
INTEGER. Array, size at least max(1, *n*).
The array *ipiv*, as returned by hetrf_rook.

anorm
REAL for checon_rook
DOUBLE PRECISION for zhecon_rook.
The 1-norm of the original matrix *A* (see *Description*).

Output Parameters

rcond
REAL for checon_rook
DOUBLE PRECISION for zhecon_rook.
The reciprocal of the condition number of the matrix *A*, computed as *rcond* = 1/(*anorm* * *ainvnm*), where *ainvnm* is an estimate of the 1-norm of A^{-1} computed in this routine.

info
INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hecon_rook interface are as follows:

a
Holds the matrix *A* of size (*n*, *n*).

ipiv	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?hecon_3

Estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian matrix A.

```
call checon_3 (uplo, n, A, lda, e, ipiv, anorm, rcond, work, info)
call zhecon_3(uplo, n, A, lda, e, ipiv, anorm, rcond, work, info)
```

Description

?hecon_3 estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian matrix A using the factorization computed by ?hetrf_rk: A = P*U*D*(U^H)*(P^T) or A = P*L*D*(L^H)*(P^T), where U (or L) is unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as *rcond* = 1 / (*anorm* * norm(inv(A))).

This routine uses BLAS3 solver ?hetrs_3.

Input Parameters

uplo	CHARACTER*1. Specifies whether the details of the factorization are stored as an upper or lower triangular matrix: = 'U': Upper triangular, form is A = P*U*D*(U^H)*(P^T); = 'L': Lower triangular, form is A = P*L*D*(L^H)*(P^T).
n	INTEGER. The order of the matrix A. $n \geq 0$.
A	COMPLEX for checon_3 COMPLEX*16 for zhecon_3 Array, dimension (*lda*,*n*). Diagonal of the block diagonal matrix D and factor U or L as computed by ?hetrf_rk: • *Only* diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A—that is, D(*k*,*k*) = A(*k*,*k*). Superdiagonal (or subdiagonal) elements of D must be provided on entry in array *e*. *—and—* • If *uplo* = 'U', factor U in the superdiagonal part of A. If *uplo* = 'L', factor L in the subdiagonal part of A.
lda	INTEGER The leading dimension of the array *A*.*lda* ≥ max(1, *n*).
e	COMPLEX for checon_3 COMPLEX*16 for zhecon_3

Array, dimension (*n*). On entry, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', e(i) = D(*i*–1, *i*),*i*=2:N, and e(1) is not referenced. If *uplo* = 'L', e(*i*) = D(*i*+1,*i*), *i*=1:N–1, and e(*n*) is not referenced.

NOTE For 1-by-1 diagonal block D(*k*), where 1 ≤ *k* ≤ *n*, the element *e*(*k*) is not referenced in both the *uplo* = 'U' and *uplo* = 'L' cases.

ipiv INTEGER

Array, dimension (*n*). Details of the interchanges and the block structure of D as determined by ?hetrf_rk.

anorm COMPLEX for checon_3

COMPLEX*16 for zhecon_3

The 1-norm of the original matrix A.

Output Parameters

rcond COMPLEX for checon_3

COMPLEX*16 for zhecon_3

The reciprocal of the condition number of the matrix A, computed as *rcond* = 1/(*anorm* * *AINVNM*), where *AINVNM* is an estimate of the 1-norm of inv(A) computed in this routine.

work COMPLEX for checon_3

COMPLEX*16 for zhecon_3

Array, dimension (2**n*).

info INTEGER.

- = 0: Successful exit.
- < 0: If *info* = -*i*, the *i*th argument had an illegal value.

?spcon

Estimates the reciprocal of a packed symmetric matrix.

Syntax

```
call sspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call dspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call cspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call spcon( ap, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a packed symmetric matrix *A*:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is symmetric, $\kappa_\infty(A) = \kappa_1(A)$).

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as *rcond* = 1 / (||*A*|| ||A^{-1}||).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?sptrf to compute the factorization of *A*.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *ap* stores the packed upper triangular factor *U* of the factorization *A* = $U*D*U^T$.

If *uplo* = 'L', the array *ap* stores the packed lower triangular factor *L* of the factorization *A* = $L*D*L^T$.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

ap, *work*

REAL for sspcon

DOUBLE PRECISION for dspcon

COMPLEX for cspcon

DOUBLE COMPLEX for zspcon.

Arrays: *ap*(*), *work*(*).

The array *ap* contains the packed factored matrix *A*, as returned by ?sptrf. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2).

The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 2*n).

ipiv

INTEGER. Array, size at least max(1, *n*).

The array *ipiv*, as returned by ?sptrf.

anorm

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix *A* (see *Description*).

iwork

INTEGER. Workspace array, size at least max(1, *n*).

Output Parameters

rcond

REAL for single precision flavors.

	`DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `spcon` interface are as follows:

ap	Holds the array *A* of size `(n*(n+1)/2)`.
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?hpcon
Estimates the reciprocal of the condition number of a packed Hermitian matrix.

Syntax

```
call chpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zhpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call hpcon( ap, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix *A*:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is Hermitian, $\kappa_\infty(A) = k_1(A)$).

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as *rcond* = 1 / (||*A*|| $||A^{-1}||$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call `?hptrf` to compute the factorization of *A*.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates how the input matrix *A* has been factored:

If *uplo* = `'U'`, the array *ap* stores the packed upper triangular factor *U* of the factorization $A = U*D*U^T$.

If *uplo* = `'L'`, the array *ap* stores the packed lower triangular factor *L* of the factorization $A = L*D*L^T$.

n

`INTEGER`. The order of matrix *A*; $n \geq 0$.

ap, *work*

`COMPLEX` for `chpcon`

`DOUBLE COMPLEX` for `zhpcon`.

The array *ap*`(*)` contains the packed factored matrix *A*, as returned by `?hptrf`. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2).

The array *work*`(*)` is a workspace for the routine. The dimension of *work* must be at least `max(1, 2*n)`.

ipiv

`INTEGER`.

Array, size at least `max(1, n)`. The array *ipiv*, as returned by `?hptrf`.

anorm

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

The norm of the *original* matrix *A* (see *Description*).

Output Parameters

rcond

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

`INTEGER`. If *info* = `0`, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hbcon` interface are as follows:

ap	Holds the array *A* of size `(n*(n+1)/2)`.
ipiv	Holds the vector of length *n*.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations `A*x = b`; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

See Also

Matrix Storage Schemes

?trcon
Estimates the reciprocal of the condition number of a triangular matrix.

Syntax

```
call strcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call dtrcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call ctrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
call ztrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
call trcon( a, rcond [,uplo] [,diag] [,norm] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a triangular matrix *A* in either the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1 \, ||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$

$\kappa_\infty(A) = ||A||_\infty \, ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H)$.

Input Parameters

norm	`CHARACTER*1`. Must be `'1'` or `'O'` or `'I'`. If *norm* = `'1'` or `'O'`, then the routine estimates the condition number of matrix *A* in 1-norm. If `norm = 'I'`, then the routine estimates the condition number of matrix *A* in infinity-norm.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates whether *A* is upper or lower triangular: If `uplo = 'U'`, the array *a* stores the upper triangle of *A*, other array elements are not referenced.

If *uplo* = 'L', the array *a* stores the lower triangle of *A*, other array elements are not referenced.

diag — CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.

If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *a*.

n — INTEGER. The order of the matrix *A*; $n \geq 0$.

a, *work* — REAL for strcon

DOUBLE PRECISION for dtrcon

COMPLEX for ctrcon

DOUBLE COMPLEX for ztrcon.

The array *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1,*n*).

The array *work*(*) is a workspace for the routine. The dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

lda — INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).

iwork — INTEGER. Workspace array, size at least max(1, *n*).

rwork — REAL for ctrcon

DOUBLE PRECISION for ztrcon.

Workspace array, size at least max(1, *n*).

Output Parameters

rcond — REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine trcon interface are as follows:

a — Holds the matrix *A* of size (*n*, *n*).

norm	Must be '1', 'O', or 'I'. The default value is '1'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A*x = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

See Also

Matrix Storage Schemes

?tpcon
Estimates the reciprocal of the condition number of a packed triangular matrix.

Syntax

```
call stpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call dtpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call ctpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
call ztpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
call tpcon( ap, rcond [,uplo] [,diag] [,norm] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a packed triangular matrix *A* in either the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1 \, ||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$

$\kappa_\infty(A) = ||A||_\infty \, ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H)$.

Input Parameters

norm	CHARACTER*1. Must be '1' or 'O' or 'I'. If *norm* = '1' or 'O', then the routine estimates the condition number of matrix *A* in 1-norm. If *norm* = 'I', then the routine estimates the condition number of matrix *A* in infinity-norm.
uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether *A* is upper or lower triangular: If *uplo* = 'U', the array *ap* stores the upper triangle of *A* in packed form.

If *uplo* = 'L', the array *ap* stores the lower triangle of *A* in packed form.

diag

CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.

If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ap*.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

ap, *work*

REAL for stpcon

DOUBLE PRECISION for dtpcon

COMPLEX for ctpcon

DOUBLE COMPLEX for ztpcon.

The array *ap*(*) contains the packed matrix *A*. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2).

The array *work*(*) is a workspace for the routine. The dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

iwork

INTEGER. Workspace array, size at least max(1, *n*).

rwork

REAL for ctpcon

DOUBLE PRECISION for ztpcon.

Workspace array, size at least max(1, *n*).

Output Parameters

rcond

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine tpcon interface are as follows:

ap

Holds the array *A* of size (*n**(*n*+1)/2).

norm

Must be '1', 'O', or 'I'. The default value is '1'.

uplo	Must be 'U' or 'L'. The default value is 'U'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A*x = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

See Also

Matrix Storage Schemes

?tbcon
Estimates the reciprocal of the condition number of a triangular band matrix.

Syntax

```
call stbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call dtbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call ctbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
call ztbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
call tbcon( ab, rcond [,uplo] [,diag] [,norm] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a triangular band matrix *A* in either the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1 \, ||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$

$\kappa_\infty(A) = ||A||_\infty \, ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H)$.

Input Parameters

norm	CHARACTER*1. Must be '1' or 'O' or 'I'. If *norm* = '1' or 'O', then the routine estimates the condition number of matrix *A* in 1-norm. If *norm* = 'I', then the routine estimates the condition number of matrix *A* in infinity-norm.
uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether *A* is upper or lower triangular: If *uplo* = 'U', the array *ap* stores the upper triangle of *A* in packed form.

If *uplo* = 'L', the array *ap* stores the lower triangle of *A* in packed form.

diag

CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.

If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ab*.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

kd

INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.

ab, *work*

REAL for stbcon

DOUBLE PRECISION for dtbcon

COMPLEX for ctbcon

DOUBLE COMPLEX for ztbcon.

The array *ab*(*ldab*,*) contains the band matrix *A*. The second dimension of *ab* must be at least max(1,*n*).

The array *work* is a workspace for the routine. The dimension of *work*(*) must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

ldab

INTEGER. The leading dimension of the array *ab*. ($ldab \geq kd + 1$).

iwork

INTEGER. Workspace array, size at least max(1, *n*).

rwork

REAL for ctbcon

DOUBLE PRECISION for ztbcon.

Workspace array, size at least max(1, *n*).

Output Parameters

rcond

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine tbcon interface are as follows:

ab	Holds the array *A* of size `(kd+1,n)`.
norm	Must be `'1'`, `'O'`, or `'I'`. The default value is `'1'`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
diag	Must be `'N'` or `'U'`. The default value is `'N'`.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations `A*x = b`; the number is usually 4 or 5 and never more than 11. Each solution requires approximately `2*n(kd + 1)` floating-point operations for real flavors and `8*n(kd + 1)` operations for complex flavors.

See Also

Matrix Storage Schemes

Refining the Solution and Estimating Its Error: LAPACK Computational Routines

This section describes the LAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see Routines for Matrix Factorization and Routines for Solving Systems of Linear Equations).

?gerfs
Refines the solution of a system of linear equations with a general coefficient matrix and estimates its error.

Syntax

```
call sgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call cgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call gerfs( a, af, ipiv, b, x [,trans] [,ferr] [,berr] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine performs an iterative refinement of the solution to a system of linear equations `A*X = B` or A^T`*X = B` or A^H`*X = B` with a general matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| \le \beta |a_{ij}|$, $|\delta b_i| \le \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?getrf
- call the solver routine ?getrs.

Input Parameters

trans

CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', the system has the form *A*X = B*.

If *trans* = 'T', the system has the form $A^T*X = B$.

If *trans* = 'C', the system has the form $A^H*X = B$.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a,*af*,*b*,*x*,*work*

REAL for sgerfs

DOUBLE PRECISION for dgerfs

COMPLEX for cgerfs

DOUBLE COMPLEX for zgerfs.

Arrays:

a(size *lda* by *) contains the original matrix *A*, as supplied to ?getrf.

af(size *ldaf* by *) contains the factored matrix *A*, as returned by ?getrf.

b(size *ldb* by *) contains the right-hand side matrix *B*.

x(size *ldx* by *) contains the solution matrix *X*.

work(size *) is a workspace array.

The second dimension of *a* and *af* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

lda

INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf

INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ldb

INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ldx

INTEGER. The leading dimension of *x*; $ldx \geq \max(1, n)$.

ipiv

INTEGER.

Array, size at least max(1, *n*).

The *ipiv* array, as returned by ?getrf.

iwork

INTEGER.

	Workspace array, size at least `max(1, n)`.
rwork	`REAL` for `cgerfs`
	`DOUBLE PRECISION` for `zgerfs`.
	Workspace array, size at least `max(1, n)`.

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	`REAL` for single precision flavors
	`DOUBLE PRECISION` for double precision flavors.
	Arrays, size at least `max(1, nrhs)`. Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	`INTEGER`. If *info* = 0, the execution is successful.
	If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gerfs` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
af	Holds the matrix *AF* of size (*n*, *n*).
ipiv	Holds the vector of length *n*.
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$ with the same coefficient matrix *A* and different right hand sides *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?gerfsx
Uses extra precise iterative refinement to improve the solution to the system of linear equations with a general coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
call sgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
call dgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
call cgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
call zgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed*, *r*, and *c* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form $A*X = B$ (No transpose).

If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose).

If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate transpose for complex flavors, Transpose for real flavors).

equed

CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

Specifies the form of equilibration that was done to *A* before calling this routine.

If *equed* = 'N', no equilibration was done.

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

If *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag(r)*A*diag(c)*. The right-hand side *B* has been changed accordingly.

n — INTEGER. The number of linear equations; the order of the matrix *A*; *n*≥ 0.

nrhs — INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; *nrhs*≥ 0.

a, *af*, *b*, *work* — REAL for sgerfsx

DOUBLE PRECISION for dgerfsx

COMPLEX for cgerfsx

DOUBLE COMPLEX for zgerfsx.

Arrays: *a* (size *lda* by *), *af* (size *ldaf* by *), *b*(size *ldb* by *), *work*(*).

The array *a* contains the original *n*-by-*n* matrix *A*.

The array *af* contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization *A = P*L*U* as computed by ?getrf.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work (size *) is a workspace array. The dimension of *work* must be at least max(1,4**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

lda — INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).

ldaf — INTEGER. The leading dimension of *af*; *ldaf*≥ max(1, *n*).

ipiv — INTEGER.

Array, size at least max(1, *n*). Contains the pivot indices as computed by ?getrf; for row 1 ≤*i*≤*n*, row *i* of the matrix was interchanged with row *ipiv*(*i*).

r, *c* — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r* (size *n*), *c* (size *n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*.

equed = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.

If *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.

If *equed* = 'C' or 'B', each element of *c* must be positive.

Each element of *r* or *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

x

REAL for sgerfsx

DOUBLE PRECISION for dgerfsx

COMPLEX for cgerfsx

DOUBLE COMPLEX for zgerfsx.

Array, size *ldx* by *.

The solution matrix *X* as computed by ?getrs

ldx

INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).

n_err_bnds

INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

nparams

INTEGER. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10.0

Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork — INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, 3**n*); used in complex flavors only.

Output Parameters

x — REAL for sgerfsx

DOUBLE PRECISION for dgerfsx

COMPLEX for cgerfsx

DOUBLE COMPLEX for zgerfsx.

The improved solution matrix *X*.

rcond — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the componentwise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

err_bnds_norm — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left|Xtrue_{ji} - X_{ji}\right|}{\max_j \left|X_{ji}\right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the *i*-th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors.
err=2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are: $\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$ Let *z=s*a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp — `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.
err=2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are: $\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$

Let *z*=*s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in *params(1)*, *params(2)*, *params(3)*. In such a case, the corresponding elements of *params* are filled with default values on output.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If 0 < *info*≤*n*: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params(3)* = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that *err_bnds_norm*(*j*,1) = 0.0 or *err_bnds_comp*(*j*,1) = 0.0). See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?gbrfs
Refines the solution of a system of linear equations with a general band coefficient matrix and estimates its error.

Syntax

```
call sgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
```

```
call dgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
```

```
call cgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

```
call zgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

```
call gbrfs( ab, afb, ipiv, b, x [,kl] [,trans] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a band matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?gbtrf
- call the solver routine ?gbtrs.

Input Parameters

trans

CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', the system has the form $A*X = B$.

If *trans* = 'T', the system has the form $A^T*X = B$.

If *trans* = 'C', the system has the form $A^H*X = B$.

n

INTEGER. The order of the matrix A; $n \geq 0$.

kl

INTEGER. The number of sub-diagonals within the band of A; $kl \geq 0$.

ku

INTEGER. The number of super-diagonals within the band of A; $ku \geq 0$.

nrhs

INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ab,*afb*,*b*,*x*,*work*

REAL for sgbrfs

DOUBLE PRECISION for dgbrfs

COMPLEX for cgbrfs

DOUBLE COMPLEX for zgbrfs.

Arrays:

ab(size *ldab* by *) contains the original band matrix A, as supplied to ?gbtrf, but stored in rows from 1 to $kl + ku + 1$.

afb(size *ldafb* by *) contains the factored band matrix A, as returned by ?gbtrf.

b(size *ldb* by *) contains the right-hand side matrix B.

x(size *ldx* by *) contains the solution matrix X.

work(*) is a workspace array.

The second dimension of *ab* and *afb* must be at least `max(1, n)`; the second dimension of *b* and *x* must be at least `max(1, nrhs)`; the dimension of *work* must be at least `max(1, 3*n)` for real flavors and `max(1, 2*n)` for complex flavors.

ldab — `INTEGER`. The leading dimension of *ab*.

ldafb — `INTEGER`. The leading dimension of *afb*.

ldb — `INTEGER`. The leading dimension of *b*; *ldb*≥ `max(1, n)`.

ldx — `INTEGER`. The leading dimension of *x*; *ldx*≥ `max(1, n)`.

ipiv — `INTEGER`.

Array, size at least `max(1, n)`. The *ipiv* array, as returned by `?gbtrf`.

iwork — `INTEGER`. Workspace array, size at least `max(1, n)`.

rwork — `REAL` for `cgbrfs`

`DOUBLE PRECISION` for `zgbrfs`.

Workspace array, size at least `max(1, n)`.

Output Parameters

x — The refined solution matrix *X*.

ferr, *berr* — `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Arrays, size at least `max(1, nrhs)`. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — `INTEGER`. If *info* =0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gbrfs` interface are as follows:

ab — Holds the array *A* of size (*kl*+*ku*+1,*n*).

afb — Holds the array *AF* of size (2**kl**ku*+1,*n*).

ipiv — Holds the vector of length *n*.

b — Holds the matrix *B* of size (*n*, *nrhs*).

x — Holds the matrix *X* of size (*n*, *nrhs*).

ferr — Holds the vector of length (*nrhs*).

berr — Holds the vector of length (*nrhs*).

trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *ku* = *lda*-*kl*-1.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of 4*n*(*kl* + *ku*) floating-point operations (for real flavors) or 16*n*(*kl* + *ku*) operations (for complex flavors). In addition, each step of iterative refinement involves 2*n*(4*kl* + 3*ku*) operations (for real flavors) or 8*n*(4*kl* + 3*ku*) operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?gbrfsx
Uses extra precise iterative refinement to improve the solution to the system of linear equations with a banded coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
call sgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb, x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
```

```
call dgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb, x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
```

```
call cgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb, x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

```
call zgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb, x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed*, *r*, and *c* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form *A***X* = *B* (No transpose).

If *trans* = 'T', the system has the form A^T**X* = *B* (Transpose).

If *trans* = 'C', the system has the form A^H**X* = *B* (Conjugate transpose for complex flavors, Transpose for real flavors).

equed

CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

Specifies the form of equilibration that was done to *A* before calling this routine.

If *equed* = 'N', no equilibration was done.

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

If *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*r*)**A***diag*(*c*). The right-hand side *B* has been changed accordingly.

n

INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

kl

INTEGER. The number of subdiagonals within the band of *A*; $kl \geq 0$.

ku

INTEGER. The number of superdiagonals within the band of *A*; $ku \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

ab, *afb*, *b*, *work*

REAL for sgbrfsx

DOUBLE PRECISION for dgbrfsx

COMPLEX for cgbrfsx

DOUBLE COMPLEX for zgbrfsx.

Arrays: *ab*(*ldab*,*), *afb*(*ldafb*,*), *b*(*ldb*,*), *work*(*).

The array *ab* contains the original matrix *A* in band storage, in rows 1 to *kl* +*ku*+1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

ab(*ku*+1+*i*-*j*,*j*) = *A*(*i*,*j*) for max(1,*j*-*ku*) $\leq i \leq$ min(*n*,*j*+*kl*).

The array *afb* contains details of the LU factorization of the banded matrix *A* as computed by ?gbtrf. *U* is stored as an upper triangular banded matrix with *kl* + *ku* superdiagonals in rows 1 to *kl* + *ku* + 1. The multipliers used during the factorization are stored in rows *kl* + *ku* + 2 to 2**kl* + *ku* + 1.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,4**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

ldab — INTEGER. The leading dimension of the array *ab*; *ldab*≥*kl*+*ku*+1.

ldafb — INTEGER. The leading dimension of the array *afb*; *ldafb*≥ 2**kl*+*ku*+1.

ipiv — INTEGER.

Array, size at least max(1, *n*). Contains the pivot indices as computed by ?gbtrf; for row 1 ≤*i*≤*n*, row *i* of the matrix was interchanged with row *ipiv*(*i*).

r, *c* — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r*(*n*), *c*(*n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*.

If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.

If *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.

If *equed* = 'C' or 'B', each element of *c* must be positive.

Each element of *r* or *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb — INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

x — REAL for sgbrfsx

DOUBLE PRECISION for dgbrfsx

COMPLEX for cgbrfsx

DOUBLE COMPLEX for zgbrfsx.

Array, size (*ldx*,*).

The solution matrix *X* as computed by sgbtrs/dgbtrs for real flavors or cgbtrs/zgbtrs for complex flavors.

ldx — INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).

n_err_bnds — INTEGER. Number of error bounds to return for each right-hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

nparams — INTEGER. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.

params — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10.0
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, 3*n); used in complex flavors only.

Output Parameters

x REAL for sgbrfsx

DOUBLE PRECISION for dgbrfsx

COMPLEX for cgbrfsx

DOUBLE COMPLEX for zgbrfsx.

The improved solution matrix *X*.

rcond REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the componentwise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

err_bnds_norm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left| Xtrue_{ji} - X_{ji} \right|}{\max_j \left| X_{ji} \right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm*(*i*,:) corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm*(:,*err*) contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision

flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let *z=s*a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|Xtrue_{ji} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.

err=2

"Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold

`sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let `z=s*(a*diag(x))`, where *x* is the solution for the current right-hand side and *s* scales each row of `a*diag(x)` by a power of the radix so all absolute row sums of *z* are approximately 1.

params

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in `params(1)`, `params(2)`, and `params(3)`. In such a case, the corresponding elements of `params` are filled with default values on output.

info

`INTEGER`. If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, the *i*-th parameter had an illegal value.

If 0 < `info`≤*n*: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; `rcond` = 0 is returned.

If `info = n+j`: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params(3) = 0.0`, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that `err_bnds_norm(j,1) = 0.0` or `err_bnds_comp(j,1) = 0.0`. See the definition of `err_bnds_norm` and `err_bnds_comp` for *err* = 1. To get information about all of the right-hand sides, check `err_bnds_norm` or `err_bnds_comp`.

See Also

Matrix Storage Schemes

?gtrfs

Refines the solution of a system of linear equations with a tridiagonal coefficient matrix and estimates its error.

Syntax

```
call sgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
```

```
call dgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
```

```
call cgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )

call zgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )

call gtrfs( dl, d, du, dlf, df, duf, du2, ipiv, b, x [,trans] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a tridiagonal matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \leq \beta|a_{ij}|$, $|\delta b_i|/|b_i| \leq \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?gttrf
- call the solver routine ?gttrs.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', the system has the form $A*X = B$. If *trans* = 'T', the system has the form $A^T*X = B$. If *trans* = 'C', the system has the form $A^H*X = B$.
n	INTEGER. The order of the matrix A; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B; $nrhs \geq 0$.
dl,*d*,*du*,*dlf*, *df*,*duf*,*du2*, *b*,*x*,*work*	REAL for sgtrfs DOUBLE PRECISION for dgtrfs COMPLEX for cgtrfs DOUBLE COMPLEX for zgtrfs. Arrays: *dl*, dimension (*n* -1), contains the subdiagonal elements of A. *d*, dimension (*n*), contains the diagonal elements of A. *du*, dimension (*n* -1), contains the superdiagonal elements of A. *dlf*, dimension (*n* - 1), contains the (*n* - 1) multipliers that define the matrix L from the LU factorization of A as computed by ?gttrf.

df, dimension (*n*), contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

duf, dimension (*n* -1), contains the (*n* - 1) elements of the first superdiagonal of *U*.

du2, dimension (*n* -2), contains the (*n* - 2) elements of the second superdiagonal of *U*.

`b(ldb,nrhs)` contains the right-hand side matrix *B*.

`x(ldx,nrhs)` contains the solution matrix *X*, as computed by `?gttrs`.

`work(*)` is a workspace array; the dimension of *work* must be at least `max(1, 3*n)` for real flavors and `max(1, 2*n)` for complex flavors.

ldb — `INTEGER`. The leading dimension of *b*; *ldb*≥ `max(1, n)`.

ldx — `INTEGER`. The leading dimension of *x*; *ldx*≥ `max(1, n)`.

ipiv — `INTEGER`.

Array, size at least `max(1, n)`. The *ipiv* array, as returned by `?gttrf`.

iwork — `INTEGER`. Workspace array, size (*n*). Used for real flavors only.

rwork — `REAL` for `cgtrfs`

`DOUBLE PRECISION` for `zgtrfs`.

Workspace array, size (*n*). Used for complex flavors only.

Output Parameters

x — The refined solution matrix *X*.

ferr, *berr* — `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Arrays, size at least `max(1,nrhs)`. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — `INTEGER`. If *info* = `0`, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gtrfs` interface are as follows:

dl — Holds the vector of length (*n*-1).

d — Holds the vector of length *n*.

du — Holds the vector of length (*n*-1).

dlf — Holds the vector of length (*n*-1).

df — Holds the vector of length *n*.

duf	Holds the vector of length (*n*-1).
du2	Holds the vector of length (*n*-2).
ipiv	Holds the vector of length *n*.
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.

See Also

Matrix Storage Schemes

?porfs
Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite coefficient matrix and estimates its error.

Syntax

```
call sporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call cporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call porfs( a, af, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations *A*X = B* with a symmetric (Hermitian) positive definite matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?potrf
- call the solver routine ?potrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
a,*af*,*b*,*x*,*work*	REAL for sporfs DOUBLE PRECISION for dporfs COMPLEX for cporfs DOUBLE COMPLEX for zporfs. Arrays: *a*(*lda*,*) contains the original matrix *A*, as supplied to ?potrf. *af*(*ldaf*,*) contains the factored matrix *A*, as returned by ?potrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The second dimension of *a* and *af* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.
lda	INTEGER. The leading dimension of *a*; $lda \geq$ max(1, *n*).
ldaf	INTEGER. The leading dimension of *af*; $ldaf \geq$ max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; $ldb \geq$ max(1, *n*).
ldx	INTEGER. The leading dimension of *x*; $ldx \geq$ max(1, *n*).
iwork	INTEGER. Workspace array, size at least max(1, *n*).
rwork	REAL for cporfs DOUBLE PRECISION for zporfs. Workspace array, size at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, size at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine porfs interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
af	Holds the matrix *AF* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations *A*x = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?porfsx
Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric/Hermitian positive-definite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
call sporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, ldx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )

call dporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, ldx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )

call cporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, ldx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )

call zporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, ldx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed* and *s* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

equed

CHARACTER*1. Must be 'N' or 'Y'.

Specifies the form of equilibration that was done to *A* before calling this routine.

If *equed* = 'N', no equilibration was done.

If *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by *diag(s)*A*diag(s)*. The right-hand side *B* has been changed accordingly.

n

INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

a, *af*, *b*, *work*

REAL for sporfsx

DOUBLE PRECISION for dporfsx

COMPLEX for cporfsx

DOUBLE COMPLEX for zporfsx.

Arrays: *a(lda,*)*, *af(ldaf,*)*, *b(ldb,*)*, *work(*)*.

The array *a* contains the symmetric/Hermitian matrix *A* as specified by *uplo*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A* and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A* and the strictly upper triangular part of *a* is not referenced. The second dimension of *a* must be at least max(1,*n*).

The array *af* contains the triangular factor *L* or *U* from the Cholesky factorization *A = U**T*U* or *A = L*L**T* as computed by spotrf for real flavors or dpotrf for complex flavors.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least `max(1,nrhs)`.

work(*) is a workspace array. The dimension of *work* must be at least `max(1,4*n)` for real flavors, and at least `max(1,2*n)` for complex flavors.

lda — `INTEGER`. The leading dimension of *a*; *lda* ≥ `max(1, n)`.

ldaf — `INTEGER`. The leading dimension of *af*; *ldaf* ≥ `max(1, n)`.

s — `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array of size *n*. The array *s* contains the scale factors for *A*.

If *equed* = `'N'`, *s* is not accessed.

If *equed* = `'Y'`, each element of *s* must be positive.

Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb — `INTEGER`. The leading dimension of the array *b*; *ldb* ≥ `max(1, n)`.

x — `REAL` for `sporfsx`

`DOUBLE PRECISION` for `dporfsx`

`COMPLEX` for `cporfsx`

`DOUBLE COMPLEX` for `zporfsx`.

Array, size *ldx* by *.

The solution matrix *X* as computed by `?potrs`

ldx — `INTEGER`. The leading dimension of the output array *x*; *ldx* ≥ `max(1, n)`.

n_err_bnds — `INTEGER`. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

nparams — `INTEGER`. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.

params — `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params`(1)` : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10.0
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, 3**n*); used in complex flavors only.

Output Parameters

x REAL for sporfsx

DOUBLE PRECISION for dporfsx

COMPLEX for cporfsx

DOUBLE COMPLEX for zporfsx.

The improved solution matrix *X*.

rcond REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least `max(1, nrhs)`. Contains the componentwise relative backward error for each solution vector `x(j)`, that is, the smallest relative change in any element of *A* or *B* that makes `x(j)` an exact solution.

err_bnds_norm

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left| Xtrue_{ji} - X_{ji} \right|}{\max_j \left| X_{ji} \right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the *i*-th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.

err=2

"Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are

$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$

Let `z=s*(a*diag(x))`, where *x* is the solution for the current right-hand side and *s* scales each row of `a*diag(x)` by a power of the radix so all absolute row sums of *z* are approximately 1.

params

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Output parameter only if the input contains erroneous values, namely in `params(1)`, `params(2)`, or `params(3)`. In such a case, the corresponding elements of `params` are filled with default values on output.

info

`INTEGER`. If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, the *i*-th parameter had an illegal value.

If 0 < `info`≤*n*: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; `rcond` = 0 is returned.

If `info = n+j`: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params(3) = 0.0`, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that `err_bnds_norm(j,1) = 0.0` or `err_bnds_comp(j,1) = 0.0`. See the definition of `err_bnds_norm` and `err_bnds_comp` for *err* = 1. To get information about all of the right-hand sides, check `err_bnds_norm` or `err_bnds_comp`.

See Also

Matrix Storage Schemes

?pprfs
Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite coefficient matrix stored in a packed format and estimates its error.

Syntax

```
call spprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call cpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call pprfs( ap, afp, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine performs an iterative refinement of the solution to a system of linear equations `A*X = B` with a symmetric (Hermitian) positive definite matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution

$||x - x_e||_\infty / ||x||_\infty$

where x_e is the exact solution.

Before calling this routine:

- call the factorization routine `?pptrf`
- call the solver routine `?pptrs`.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates how the input matrix *A* has been factored:

If `uplo = 'U'`, the upper triangle of *A* is stored.

If `uplo = 'L'`, the lower triangle of *A* is stored.

n

`INTEGER`. The order of the matrix *A*; $n \geq 0$.

nrhs

`INTEGER`. The number of right-hand sides; $nrhs \geq 0$.

ap, *afp*, *b*, *x*, *work*

`REAL` for `spprfs`

`DOUBLE PRECISION` for `dpprfs`

`COMPLEX` for `cpprfs`

`DOUBLE COMPLEX` for `zpprfs`.

Arrays:

`ap(*)` contains the original matrix *A* in a packed format, as supplied to `?pptrf`.

`afp(*)` contains the factored matrix *A* in a packed format, as returned by `?pptrf`.

`b(ldb,*)` contains the right-hand side matrix *B*.

`x(ldx,*)` contains the solution matrix *X*.

`work(*)` is a workspace array.

The dimension of arrays *ap* and *afp* must be at least `max(1,n(n+1)/2)`; the second dimension of *b* and *x* must be at least `max(1, nrhs)`; the dimension of *work* must be at least `max(1, 3*n)` for real flavors and `max(1, 2*n)` for complex flavors.

ldb

`INTEGER`. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ldx

`INTEGER`. The leading dimension of *x*; $ldx \geq \max(1, n)$.

iwork	INTEGER. Workspace array, size at least max(1, *n*).
rwork	REAL for cpprfs DOUBLE PRECISION for zpprfs. Workspace array, size at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, size at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pprfs interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
afp	Holds the array *AF* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations *A***x* = *b*; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?pbrfs
Refines the solution of a system of linear equations with a band symmetric (Hermitian) positive-definite coefficient matrix and estimates its error.

Syntax

```
call spbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call cpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call pbrfs( ab, afb, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ with a symmetric (Hermitian) positive definite band matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?pbtrf
- call the solver routine ?pbtrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If *uplo* = 'U', the upper triangle of A is stored. If *uplo* = 'L', the lower triangle of A is stored.
n	INTEGER. The order of the matrix A; $n \geq 0$.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix A; $kd \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ab,*afb*,*b*,*x*,*work*	REAL for spbrfs DOUBLE PRECISION for dpbrfs

COMPLEX for cpbrfs

DOUBLE COMPLEX for zpbrfs.

Arrays:

ab(*ldab*,*) contains the original band matrix *A*, as supplied to ?pbtrf.

afb(*ldafb*,*) contains the factored band matrix *A*, as returned by ?pbtrf.

b(*ldb*,*) contains the right-hand side matrix *B*.

x(*ldx*,*) contains the solution matrix *X*.

work(*) is a workspace array.

The second dimension of *ab* and *afb* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

ldab — INTEGER. The leading dimension of *ab*; *ldab*≥*kd* + 1.

ldafb — INTEGER. The leading dimension of *afb*; *ldafb*≥*kd* + 1.

ldb — INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

ldx — INTEGER. The leading dimension of *x*; *ldx*≥ max(1, *n*).

iwork — INTEGER. Workspace array, size at least max(1, *n*).

rwork — REAL for cpbrfs

DOUBLE PRECISION for zpbrfs.

Workspace array, size at least max(1, *n*).

Output Parameters

x — The refined solution matrix *X*.

ferr, *berr* — REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, size at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pbrfs interface are as follows:

ab — Holds the array *A* of size (*kd*+1, *n*).

afb	Holds the array *AF* of size (*kd*+1, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $8n*kd$ floating-point operations (for real flavors) or $32n*kd$ operations (for complex flavors). In addition, each step of iterative refinement involves $12n*kd$ operations (for real flavors) or $48n*kd$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4n*kd$ floating-point operations for real flavors or $16n*kd$ for complex flavors.

See Also

Matrix Storage Schemes

?ptrfs
Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal coefficient matrix and estimates its error.

Syntax

```
call sptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call dptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call cptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call ptrfs( d, df, e, ef, b, x [,ferr] [,berr] [,info] )
call ptrfs( d, df, e, ef, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ with a symmetric (Hermitian) positive definite tridiagonal matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?pttrf
- call the solver routine ?pttrs.

Input Parameters

uplo

CHARACTER*1. Used for complex flavors only. Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix *A* is stored and how *A* is factored:

If *uplo* = 'U', the array *e* stores the superdiagonal of *A*, and *A* is factored as U^H*D*U.

If *uplo* = 'L', the array *e* stores the subdiagonal of *A*, and *A* is factored as $L*D*L^H$.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; $nrhs \geq 0$.

d, *df*, *rwork*

REAL for single precision flavors DOUBLE PRECISION for double precision flavors

Arrays: *d*(*n*), *df*(*n*), *rwork*(*n*).

The array *d* contains the *n* diagonal elements of the tridiagonal matrix *A*.

The array *df* contains the *n* diagonal elements of the diagonal matrix *D* from the factorization of *A* as computed by ?pttrf.

The array *rwork* is a workspace array used for complex flavors only.

e,*ef*,*b*,*x*,*work*

REAL for sptrfs

DOUBLE PRECISION for dptrfs

COMPLEX for cptrfs

DOUBLE COMPLEX for zptrfs.

Arrays: *e*(*n* -1), *ef*(*n* -1), *b*(*ldb*,*nrhs*), *x*(*ldx*,*nrhs*), *work*(*).

The array *e* contains the (*n* - 1) off-diagonal elements of the tridiagonal matrix *A* (see *uplo*).

The array *ef* contains the (*n* - 1) off-diagonal elements of the unit bidiagonal factor *U* or *L* from the factorization computed by ?pttrf (see *uplo*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

The array *x* contains the solution matrix *X* as computed by ?pttrs.

The array *work* is a workspace array. The dimension of *work* must be at least 2*n for real flavors, and at least *n* for complex flavors.

ldb

INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ldx	INTEGER. The leading dimension of *x*; *ldx*≥ max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, size at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ptrfs interface are as follows:

d	Holds the vector of length *n*.
df	Holds the vector of length *n*.
e	Holds the vector of length (*n*-1).
ef	Holds the vector of length (*n*-1).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Used in complex flavors only. Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?syrfs
Refines the solution of a system of linear equations with a symmetric coefficient matrix and estimates its error.

Syntax

```
call ssyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
```

```
call dsyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
```

```
call csyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
rwork, info )
```

```
call zsyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
rwork, info )
```

```
call syrfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ with a symmetric full-storage matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?sytrf
- call the solver routine ?sytrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
a,*af*,*b*,*x*,*work*	REAL for ssyrfs DOUBLE PRECISION for dsyrfs COMPLEX for csyrfs DOUBLE COMPLEX for zsyrfs. Arrays: *a*(*lda*,*) contains the original matrix *A*, as supplied to ?sytrf. *af*(*ldaf*,*) contains the factored matrix *A*, as returned by ?sytrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array.

The second dimension of *a* and *af* must be at least `max(1, n)`; the second dimension of *b* and *x* must be at least `max(1, nrhs)`; the dimension of *work* must be at least `max(1, 3*n)` for real flavors and `max(1, 2*n)` for complex flavors.

lda — `INTEGER`. The leading dimension of *a*; *lda* ≥ `max(1, n)`.

ldaf — `INTEGER`. The leading dimension of *af*; *ldaf* ≥ `max(1, n)`.

ldb — `INTEGER`. The leading dimension of *b*; *ldb* ≥ `max(1, n)`.

ldx — `INTEGER`. The leading dimension of *x*; *ldx* ≥ `max(1, n)`.

ipiv — `INTEGER`.

Array, size at least `max(1, n)`. The *ipiv* array, as returned by `?sytrf`.

iwork — `INTEGER`. Workspace array, size at least `max(1, n)`.

rwork — `REAL` for `csyrfs`

`DOUBLE PRECISION` for `zsyrfs`.

Workspace array, size at least `max(1, n)`.

Output Parameters

x — The refined solution matrix *X*.

ferr, *berr* — `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Arrays, size at least `max(1, nrhs)`. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — `INTEGER`. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `syrfs` interface are as follows:

a — Holds the matrix *A* of size (*n*, *n*).

af — Holds the matrix *AF* of size (*n*, *n*).

ipiv — Holds the vector of length *n*.

b — Holds the matrix *B* of size (*n*, *nrhs*).

x — Holds the matrix *X* of size (*n*, *nrhs*).

ferr — Holds the vector of length (*nrhs*).

berr — Holds the vector of length (*nrhs*).

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?syrfsx
Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
call ssyrfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )

call dsyrfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )

call csyrfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )

call zsyrfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed* and *s* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

equed — CHARACTER*1. Must be 'N' or 'Y'.

Specifies the form of equilibration that was done to *A* before calling this routine.

If *equed* = 'N', no equilibration was done.

If *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by *diag(s)*A*diag(s)*. The right-hand side *B* has been changed accordingly.

n — INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

a, *af*, *b*, *work* — REAL for ssyrfsx

DOUBLE PRECISION for dsyrfsx

COMPLEX for csyrfsx

DOUBLE COMPLEX for zsyrfsx.

Arrays: *a(lda,*)*, *af(ldaf,*)*, *b(ldb,*)*, *work(*)*.

The array *a* contains the symmetric/Hermitian matrix *A* as specified by *uplo*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A* and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A* and the strictly upper triangular part of *a* is not referenced. The second dimension of *a* must be at least max(1,*n*).

The array *af* contains the triangular factor *L* or *U* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$ as computed by ssytrf for real flavors or dsytrf for complex flavors.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,4**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

lda — INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ldaf — INTEGER. The leading dimension of *af*; *ldaf* ≥ max(1, *n*).

ipiv — INTEGER.

Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D* as determined by ssytrf for real flavors or dsytrf for complex flavors.

s — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*). The array *s* contains the scale factors for *A*.

If *equed* = 'N', *s* is not accessed.

If *equed* = 'Y', each element of *s* must be positive.

Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array *b*; *ldb* ≥ max(1, *n*).

x

REAL for ssyrfsx

DOUBLE PRECISION for dsyrfsx

COMPLEX for csyrfsx

DOUBLE COMPLEX for zsyrfsx.

Array, size *ldx* by *.

The solution matrix *X* as computed by ?sytrs

ldx

INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

n_err_bnds

INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

nparams

INTEGER. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10.0

Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork — INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, 3**n*); used in complex flavors only.

Output Parameters

x — REAL for ssyrfsx

DOUBLE PRECISION for dsyrfsx

COMPLEX for csyrfsx

DOUBLE COMPLEX for zsyrfsx.

The improved solution matrix *X*.

rcond — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the componentwise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

err_bnds_norm — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left|Xtrue_{ji} - X_{ji}\right|}{\max_j \left|X_{ji}\right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the follwoing three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.
err=2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are: $\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$

err_bnds_comp

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.
err=2	"Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are: $\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$ Let *z=s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in *params(1)*, *params(2)*, *params(3)*. In such a case, the corresponding elements of *params* are filled with default values on output.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = –*i*, the *i*-th parameter had an illegal value.

If 0 < *info*≤*n*: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params(3)* = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that *err_bnds_norm*(*j*,1) = 0.0 or *err_bnds_comp*(*j*,1) = 0.0. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?herfs
Refines the solution of a system of linear equations with a complex Hermitian coefficient matrix and estimates its error.

Syntax

```
call cherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call herfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations *A*X* = *B* with a complex Hermitian full-storage matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| \le \beta |a_{ij}|$, $|\delta b_i| \le \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?hetrf
- call the solver routine ?hetrs.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n — INTEGER. The order of the matrix *A*; *n*≥ 0.

nrhs — INTEGER. The number of right-hand sides; *nrhs*≥ 0.

a,*af*,*b*,*x*,*work* — COMPLEX for cherfs

DOUBLE COMPLEX for zherfs.

Arrays:

a(size *lda* by *) contains the original matrix *A*, as supplied to ?hetrf.

af(size *ldaf* by *) contains the factored matrix *A*, as returned by ?hetrf.

b(size *ldb* by *) contains the right-hand side matrix *B*.

x(size *ldx* by *) contains the solution matrix *X*.

work(*) is a workspace array.

The second dimension of *a* and *af* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 2**n*).

lda — INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).

ldaf — INTEGER. The leading dimension of *af*; *ldaf*≥ max(1, *n*).

ldb — INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

ldx — INTEGER. The leading dimension of *x*; *ldx*≥ max(1, *n*).

ipiv — INTEGER.

Array, size at least max(1, *n*). The *ipiv* array, as returned by ?hetrf.

rwork — REAL for cherfs

DOUBLE PRECISION for zherfs.

Workspace array, size at least max(1, *n*).

Output Parameters

x — The refined solution matrix *X*.

ferr, *berr* — REAL for cherfs

DOUBLE PRECISION for zherfs.

Arrays, size at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `herfs` interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
af	Holds the matrix *AF* of size (*n*,*n*).
ipiv	Holds the vector of length *n*.
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is `?ssyrfs`/`?dsyrfs`

See Also

Matrix Storage Schemes

?herfsx
Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
call cherfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

```
call zherfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed* and *s* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

equed

CHARACTER*1. Must be 'N' or 'Y'.

Specifies the form of equilibration that was done to *A* before calling this routine.

If *equed* = 'N', no equilibration was done.

If *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by *diag(s)*A*diag(s)*. The right-hand side *B* has been changed accordingly.

n

INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

a, *af*, *b*, *work*

COMPLEX for cherfsx

DOUBLE COMPLEX for zherfsx.

Arrays: *a(lda,*)*, *af(ldaf,*)*, *b(ldb,*)*, *work(*)*.

The array *a* contains the Hermitian matrix *A* as specified by *uplo*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A* and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A* and the strictly upper triangular part of *a* is not referenced. The second dimension of *a* must be at least max(1,*n*).

The array *af* contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ssytrf for cherfsx or dsytrf for zherfsx.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,2**n*).

lda

INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ldaf INTEGER. The leading dimension of *af*; *ldaf* ≥ max(1, *n*).

ipiv INTEGER.

Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D* as determined by ssytrf for real flavors or dsytrf for complex flavors.

s REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*). The array *s* contains the scale factors for *A*.

If *equed* = 'N', *s* is not accessed.

If *equed* = 'Y', each element of *s* must be positive.

Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array *b*; *ldb* ≥ max(1, *n*).

x COMPLEX for cherfsx

DOUBLE COMPLEX for zherfsx.

Array, size (*ldx*,*).

The solution matrix *X* as computed by ?hetrs

ldx INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

n_err_bnds INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

nparams INTEGER. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.

params REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0 (for cherfsx), 1.0D+0 (for zherfsx).

=0.0	No refinement is performed and no error bounds are computed.

=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10
Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

rwork

REAL for cherfsx

DOUBLE PRECISION for zherfsx.

Workspace array, size at least max(1, 3*n).

Output Parameters

x

COMPLEX for cherfsx

DOUBLE COMPLEX for zherfsx.

The improved solution matrix *X*.

rcond

REAL for cherfsx

DOUBLE PRECISION for zherfsx.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr

REAL for cherfsx

DOUBLE PRECISION for zherfsx.

Array, size at least max(1, *nrhs*). Contains the componentwise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

err_bnds_norm

REAL for cherfsx

DOUBLE PRECISION for zherfsx.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left|Xtrue_{ji} - X_{ji}\right|}{\max_j \left|X_{ji}\right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1 — "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for `cherfsx` and *sqrt*(*n*)**dlamch*(ε) for `zherfsx`.

err=2 — "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for `cherfsx` and *sqrt*(*n*)**dlamch*(ε) for `zherfsx`. This error bound should only be trusted if the previous boolean is true.

err=3 — Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp

`REAL` for `cherfsx`

`DOUBLE PRECISION` for `zherfsx`.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|Xtrue_{ji} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1 — "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for `cherfsx` and *sqrt*(*n*)**dlamch*(ε) for `zherfsx`.

err=2 — "Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for `cherfsx` and *sqrt*(*n*)**dlamch*(ε) for `zherfsx`. This error bound should only be trusted if the previous boolean is true.

err=3 — Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let *z=s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1.

params — `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in *params(1)*, *params(2)*, *params(3)*. In such a case, the corresponding elements of *params* are filled with default values on output.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If 0 < *info*≤*n*: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params(3)* = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that *err_bnds_norm*(*j*,1) = 0.0 or *err_bnds_comp*(*j*,1) = 0.0. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?sprfs

Refines the solution of a system of linear equations with a packed symmetric coefficient matrix and estimates the solution error.

Syntax

```
call ssprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call csprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call sprfs( ap, afp, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations *A*X* = *B* with a packed symmetric matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?sptrf
- call the solver routine ?sptrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; *n*≥ 0.
nrhs	INTEGER. The number of right-hand sides; *nrhs*≥ 0.
ap,*afp*,*b*,*x*,*work*	REAL for ssprfs DOUBLE PRECISION for dsprfs COMPLEX for csprfs DOUBLE COMPLEX for zsprfs. Arrays: *ap* (size *) contains the original packed matrix *A*, as supplied to ?sptrf. *afp* (size *) contains the factored packed matrix *A*, as returned by ?sptrf. *b*(size *ldb* by *) contains the right-hand side matrix *B*. *x*(size *ldx* by *) contains the solution matrix *X*. *work*(*) is a workspace array. The dimension of arrays *ap* and *afp* must be at least max(1, *n*(*n*+1)/2); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).
ldx	INTEGER. The leading dimension of *x*; *ldx*≥ max(1, *n*).
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?sptrf.
iwork	INTEGER. Workspace array, size at least max(1, *n*).
rwork	REAL for csprfs DOUBLE PRECISION for zsprfs. Workspace array, size at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.

ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, size at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sprfs interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
afp	Holds the array *AF* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length *n*.
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations *A***x* = *b*; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?hprfs

Refines the solution of a system of linear equations with a packed complex Hermitian coefficient matrix and estimates the solution error.

Syntax

```
call chprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
```

```
call zhprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
```

```
call hprfs( ap, afp, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations `A*X = B` with a packed complex Hermitian matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?hptrf
- call the solver routine ?hptrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; *n*≥ 0.
nrhs	INTEGER. The number of right-hand sides; *nrhs*≥ 0.
ap,*afp*,*b*,*x*,*work*	COMPLEX for chprfs DOUBLE COMPLEX for zhprfs. Arrays: *ap* (size *) contains the original packed matrix *A*, as supplied to ?hptrf. *afp* (size *) contains the factored packed matrix *A*, as returned by ?hptrf. *b*(size *ldb* by *) contains the right-hand side matrix *B*. *x*(size *ldx* by *) contains the solution matrix *X*. *work*(*) is a workspace array. The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 2*n).
ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

ldx	INTEGER. The leading dimension of *x*; *ldx*≥ max(1, *n*).
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?hptrf.
rwork	REAL for chprfs DOUBLE PRECISION for zhprfs. Workspace array, size at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for chprfs. DOUBLE PRECISION for zhprfs. Arrays, size at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hprfs interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
afp	Holds the array *AF* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length *n*.
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is ?ssprfs/?dsprfs.

See Also

Matrix Storage Schemes

?trrfs
Estimates the error in the solution of a system of linear equations with a triangular coefficient matrix.

Syntax

```
call strrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dtrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call ctrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call ztrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call trrfs( a, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the errors in the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a triangular matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

The routine also estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine ?trtrs.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether A is upper or lower triangular:

If *uplo* = 'U', then A is upper triangular.

If *uplo* = 'L', then A is lower triangular.

trans

CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', the system has the form $A*X = B$.

If *trans* = 'T', the system has the form $A^T*X = B$.

If *trans* = 'C', the system has the form $A^H*X = B$.

diag — CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.

If *diag* = 'U', then *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.

n — INTEGER. The order of the matrix *A*; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a, *b*, *x*, *work* — REAL for strrfs

DOUBLE PRECISION for dtrrfs

COMPLEX for ctrrfs

DOUBLE COMPLEX for ztrrfs.

Arrays:

a(size *lda* by *) contains the upper or lower triangular matrix *A*, as specified by *uplo*.

b(size *ldb* by *) contains the right-hand side matrix *B*.

x(size *ldx* by *) contains the solution matrix *X*.

work(*) is a workspace array.

The second dimension of *a* must be at least max(1,*n*); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1,3*n) for real flavors and max(1,2*n) for complex flavors.

lda — INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ldb — INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

ldx — INTEGER. The leading dimension of *x*; *ldx* ≥ max(1, *n*).

iwork — INTEGER. Workspace array, size at least max(1, *n*).

rwork — REAL for ctrrfs

DOUBLE PRECISION for ztrrfs.

Workspace array, size at least max(1, *n*).

Output Parameters

ferr, *berr* — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays, size at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `trrfs` interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.
diag	Must be `'N'` or `'U'`. The default value is `'N'`.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations *A*x = b*; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?tprfs
Estimates the error in the solution of a system of linear equations with a packed triangular coefficient matrix.

Syntax

```
call stprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dtprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call ctprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call ztprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call tprfs( ap, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine estimates the errors in the solution to a system of linear equations `A*X = B` or A^T`*X = B` or A^H*X = *B* with a packed triangular matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error*β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

The routine also estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine `?tptrs`.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates whether *A* is upper or lower triangular:

If `uplo = 'U'`, then *A* is upper triangular.

If `uplo = 'L'`, then *A* is lower triangular.

trans

`CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Indicates the form of the equations:

If `trans = 'N'`, the system has the form `A*X = B`.

If `trans = 'T'`, the system has the form A^T*X = *B*.

If `trans = 'C'`, the system has the form A^H*X = *B*.

diag

`CHARACTER*1`. Must be `'N'` or `'U'`.

If `diag = 'N'`, *A* is not a unit triangular matrix.

If `diag = 'U'`, *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *ap*.

n

`INTEGER`. The order of the matrix *A*; $n \geq 0$.

nrhs

`INTEGER`. The number of right-hand sides; $nrhs \geq 0$.

ap, *b*, *x*, *work*

`REAL` for `stprfs`

`DOUBLE PRECISION` for `dtprfs`

`COMPLEX` for `ctprfs`

`DOUBLE COMPLEX` for `ztprfs`.

Arrays:

ap (size *) contains the upper or lower triangular matrix *A*, as specified by *uplo*.

b(size *ldb* by *) contains the right-hand side matrix *B*.

x(size *ldx* by *) contains the solution matrix *X*.

`work(*)` is a workspace array.

	The dimension of *ap* must be at least `max(1,n(n+1)/2)`; the second dimension of *b* and *x* must be at least `max(1,nrhs)`; the dimension of *work* must be at least `max(1,3*n)` for real flavors and `max(1,2*n)` for complex flavors.
ldb	`INTEGER`. The leading dimension of *b*; *ldb*≥ `max(1, n)`.
ldx	`INTEGER`. The leading dimension of *x*; *ldx*≥ `max(1, n)`.
iwork	`INTEGER`. Workspace array, size at least `max(1, n)`.
rwork	`REAL` for `ctprfs` `DOUBLE PRECISION` for `ztprfs`. Workspace array, size at least `max(1, n)`.

Output Parameters

ferr, *berr*	`REAL` for single precision flavors `DOUBLE PRECISION` for double precision flavors. Arrays, size at least `max(1, nrhs)`. Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `tprfs` interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.
diag	Must be `'N'` or `'U'`. The default value is `'N'`.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?tbrfs
Estimates the error in the solution of a system of linear equations with a triangular band coefficient matrix.

Syntax

```
call stbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dtbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call ctbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call ztbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call tbrfs( ab, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the errors in the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a triangular band matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

The routine also estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine ?tbtrs.

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:
If *uplo* = 'U', then A is upper triangular.
If *uplo* = 'L', then A is lower triangular.

trans
CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', the system has the form $A*X = B$.

If *trans* = 'T', the system has the form $A^T*X = B$.

If *trans* = 'C', the system has the form $A^H*X = B$.

diag CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', *A* is not a unit triangular matrix.

If *diag* = 'U', *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *ab*.

n INTEGER. The order of the matrix *A*; $n \geq 0$.

kd INTEGER. The number of super-diagonals or sub-diagonals in the matrix *A*; $kd \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ab, *b*, *x*, *work* REAL for stbrfs

DOUBLE PRECISION for dtbrfs

COMPLEX for ctbrfs

DOUBLE COMPLEX for ztbrfs.

Arrays:

ab(size *ldab* by *) contains the upper or lower triangular matrix *A*, as specified by *uplo*, in band storage format.

b(size *ldb* by *) contains the right-hand side matrix *B*.

x(size *ldx* by *) contains the solution matrix *X*.

work(*) is a workspace array.

The second dimension of *a* must be at least max(1,*n*); the second dimension of *b* and *x* must be at least max(1,*nrhs*). The dimension of *work* must be at least max(1,3*n) for real flavors and max(1,2*n) for complex flavors.

ldab INTEGER. The leading dimension of the array *ab*; $ldab \geq kd + 1$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq$ max(1, *n*).

ldx INTEGER. The leading dimension of *x*; $ldx \geq$ max(1, *n*).

iwork INTEGER. Workspace array, size at least max(1, *n*).

rwork REAL for ctbrfs

DOUBLE PRECISION for ztbrfs.

Workspace array, size at least max(1, *n*).

Output Parameters

ferr, *berr* REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, size at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine tbrfs interface are as follows:

ab	Holds the array *A* of size (*kd*+1, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n*kd$ floating-point operations for real flavors or $8n*kd$ operations for complex flavors.

See Also

Matrix Storage Schemes

Matrix Inversion: LAPACK Computational Routines

It is seldom necessary to compute an explicit inverse of a matrix. In particular, do not attempt to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see Routines for Solving Systems of Linear Equations); this is more efficient and more accurate.

However, matrix inversion routines are provided for the rare occasions when an explicit inverse matrix is needed.

?getri
Computes the inverse of an LU-factored general matrix.

Syntax

```
call sgetri( n, a, lda, ipiv, work, lwork, info )
call dgetri( n, a, lda, ipiv, work, lwork, info )
call cgetri( n, a, lda, ipiv, work, lwork, info )
call zgetri( n, a, lda, ipiv, work, lwork, info )
```

```
call getri( a, ipiv [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a general matrix *A*. Before calling this routine, call ?getrf to factorize *A*.

Input Parameters

n	INTEGER. The order of the matrix *A*; $n \geq 0$.
a, *work*	REAL for sgetri DOUBLE PRECISION for dgetri COMPLEX for cgetri DOUBLE COMPLEX for zgetri. Arrays: *a*(*lda*,*), *work*(*). *a*(*lda*,*) contains the factorization of the matrix *A*, as returned by ?getrf: *A = P*L*U*. The second dimension of *a* must be at least max(1,*n*). *work*(*) is a workspace array of dimension at least max(1,*lwork*).
lda	INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?getrf.
lwork	INTEGER. The size of the *work* array; *lwork* ≥ *n*. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* below for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *n*-by-*n* matrix inv(*A*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of the factor *U* is zero, *U* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `getri` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
ipiv	Holds the vector of length *n*.

Application Notes

For better performance, try using *lwork* = *n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed inverse *X* satisfies the following error bound:

$$|XA - I| \leq c(n)\varepsilon|X|P|L||U|,$$

where $c(n)$ is a modest linear function of *n*; ε is the machine precision; *I* denotes the identity matrix; *P*, *L*, and *U* are the factors of the matrix factorization *A* = *P*L*U*.

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

mkl_?getrinp
Computes the inverse of an LU-factored general matrix without pivoting.

Syntax

```
call mkl_sgetrinp( n, a, lda, work, lwork, info )
call mkl_dgetrinp( n, a, lda, work, lwork, info )
call mkl_cgetrinp( n, a, lda, work, lwork, info )
call mkl_zgetrinp( n, a, lda, work, lwork, info )
```

Include Files

- mkl.fi

Description

The routine computes the inverse inv(*A*) of a general matrix *A*. Before calling this routine, call mkl_?getrfnp to factorize *A*.

Input Parameters

n	INTEGER. The order of the matrix *A*; *n*≥ 0.
a, *work*	REAL for mkl_sgetrinp DOUBLE PRECISION for mkl_dgetrinp COMPLEX for mkl_cgetrinp DOUBLE COMPLEX for mkl_zgetrinp. Arrays: *a*(*lda*,*), *work*(*). *a*(*lda*,*) contains the factorization of the matrix *A*, as returned by mkl_?getrfnp: *A* = *L***U*. The second dimension of *a* must be at least max(1,*n*). *work*(*) is a workspace array of dimension at least max(1,*lwork*).
lda	INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).
lwork	INTEGER. The size of the *work* array; *lwork*≥*n*. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* below for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *n*-by-*n* matrix inv(*A*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of the factor *U* is zero, *U* is singular, and the inversion could not be completed.

Application Notes

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?potri
Computes the inverse of a symmetric (Hermitian) positive-definite matrix using the Cholesky factorization.

Syntax

```
call spotri( uplo, n, a, lda, info )
call dpotri( uplo, n, a, lda, info )
call cpotri( uplo, n, a, lda, info )
call zpotri( uplo, n, a, lda, info )
call potri( a [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix *A*. Before calling this routine, call ?potrf to factorize *A*.

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the upper triangle of *A* is stored.
If *uplo* = 'L', the lower triangle of *A* is stored.

n
INTEGER. The order of the matrix *A*; *n* ≥ 0.

a
REAL for spotri
DOUBLE PRECISION for dpotri
COMPLEX for cpotri
DOUBLE COMPLEX for zpotri.
Array *a*(size *lda* by *) Contains the factorization of the matrix *A*, as returned by ?potrf.
The second dimension of *a* must be at least max(1, *n*).

lda
INTEGER. The leading dimension of a. *lda* ≥ max(1, *n*).

Output Parameters

a
Overwritten by the upper or lower triangle of the inverse of *A*.

info
INTEGER.
If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine potri interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$||XA - I||_2 \le c(n)\varepsilon\kappa_2(A), \quad ||AX - I||_2 \le c(n)\varepsilon\kappa_2(A),$$

where $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The 2-norm $||A||_2$ of a matrix *A* is defined by $||A||_2 = max_{x \cdot x=1}(Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = ||A||_2 \, ||A^{-1}||_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?pftri
Computes the inverse of a symmetric (Hermitian) positive-definite matrix in RFP format using the Cholesky factorization.

Syntax

```
call spftri( transr, uplo, n, a, info )
call dpftri( transr, uplo, n, a, info )
call cpftri( transr, uplo, n, a, info )
call zpftri( transr, uplo, n, a, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a symmetric positive definite or, for complex data, Hermitian positive-definite matrix *A* using the Cholesky factorization:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data if *uplo*='U'

$A = L*L^T$ for real data, $A = L*L^H$ for complex data if *uplo*='L'

Before calling this routine, call ?pftrf to factorize *A*.

The matrix *A* is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

Input Parameters

transr

CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data).

If *transr* = 'N', the Normal *transr* of RFP *U* (if *uplo* = 'U') or *L* (if *uplo* = 'L') is stored.

If *transr* = 'T', the Transpose *transr* of RFP *U* (if *uplo* = 'U') or *L* (if *uplo* = 'L' is stored.

If *transr* = 'C', the Conjugate-Transpose *transr* of RFP *U* (if *uplo* = 'U') or *L* (if *uplo* = 'L' is stored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', $A = U^T*U$ for real data or $A = U^H*U$ for complex data, and *U* is stored.

If *uplo* = 'L', $A = L*L^T$ for real data or $A = L*L^H$ for complex data, and *L* is stored.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

a

REAL for spftri

DOUBLE PRECISION for dpftri

COMPLEX for cpftri

DOUBLE COMPLEX for zpftri.

Array, size (*n**(*n*+1)/2). The array *a* contains the factor *U* or *L* matrix *A* in the RFP format.

Output Parameters

a

The symmetric/Hermitian inverse of the original matrix in the same storage format.

info

INTEGER. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the (*i*, *i*) element of the factor U or L is zero, and the inverse could not be computed.

See Also

Matrix Storage Schemes

?pptri

Computes the inverse of a packed symmetric (Hermitian) positive-definite matrix using Cholesky factorization.

Syntax

```
call spptri( uplo, n, ap, info )
call dpptri( uplo, n, ap, info )
call cpptri( uplo, n, ap, info )
call zpptri( uplo, n, ap, info )
call pptri( ap [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix *A* in *packed* form. Before calling this routine, call ?pptrf to factorize *A*.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular factor is stored in *ap*:

If *uplo* = 'U', then the upper triangular factor is stored.

If *uplo* = 'L', then the lower triangular factor is stored.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

ap

REAL for spptri

DOUBLE PRECISION for dpptri

COMPLEX for cpptri

DOUBLE COMPLEX for zpptri.

Array, size at least max(1, $n(n+1)/2$).

Contains the factorization of the packed matrix *A*, as returned by ?pptrf.

The dimension *ap* must be at least max(1,$n(n+1)/2$).

Output Parameters

ap

Overwritten by the packed *n*-by-*n* matrix inv(*A*).

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `pptri` interface are as follows:

ap	Holds the array *A* of size `(n*(n+1)/2)`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$||XA - I||_2 \le c(n)\varepsilon\kappa_2(A), \quad ||AX - I||_2 \le c(n)\varepsilon\kappa_2(A),$$

where $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The 2-norm $||A||_2$ of a matrix *A* is defined by $||A||_2 = max_{x \cdot x=1}(Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = ||A||_2 \, ||A^{-1}||_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?sytri
*Computes the inverse of a symmetric matrix using U*D*U^T or L*D*L^T Bunch-Kaufman factorization.*

Syntax

```
call ssytri( uplo, n, a, lda, ipiv, work, info )
call dsytri( uplo, n, a, lda, ipiv, work, info )
call csytri( uplo, n, a, lda, ipiv, work, info )
call zsytri( uplo, n, a, lda, ipiv, work, info )
call sytri( a, ipiv [,uplo] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes the inverse `inv(A)` of a symmetric matrix *A*. Before calling this routine, call `?sytrf` to factorize *A*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the array *a* stores the Bunch-Kaufman factorization $A = U*D*U^T$. If *uplo* = 'L', the array *a* stores the Bunch-Kaufman factorization $A = L*D*L^T$.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
a, *work*	REAL for ssytri DOUBLE PRECISION for dsytri COMPLEX for csytri DOUBLE COMPLEX for zsytri. Arrays: *a*(size *lda* by *) contains the factorization of the matrix *A*, as returned by ?sytrf. The second dimension of *a* must be at least max(1,*n*). *work*(*) is a workspace array. The dimension of *work* must be at least max(1,2*n).
lda	INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?sytrf.

Output Parameters

a	Overwritten by the *n*-by-*n* matrix inv(*A*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* =-*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sytri interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
ipiv	Holds the vector of length *n*.

uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^T*P^T*X*P*U - I| \le c(n)\varepsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D*L^T*P^T*X*P*L - I| \le c(n)\varepsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?sytri_rook
*Computes the inverse of a symmetric matrix using U*D*U^T or L*D*L^T bounded Bunch-Kaufman factorization.*

Syntax

```
call ssytri_rook( uplo, n, a, lda, ipiv, work, info )
call dsytri_rook( uplo, n, a, lda, ipiv, work, info )
call csytri_rook( uplo, n, a, lda, ipiv, work, info )
call zsytri_rook( uplo, n, a, lda, ipiv, work, info )
call sytri_rook( a, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a symmetric matrix *A*. Before calling this routine, call ?sytrf_rook to factorize *A*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the array *a* stores the factorization $A = U*D*U^T$. If *uplo* = 'L', the array *a* stores the factorization $A = L*D*L^T$.
n	INTEGER. The order of the matrix *A*; $n \ge 0$.
a, *work*	REAL for ssytri_rook DOUBLE PRECISION for dsytri_rook

COMPLEX for csytri_rook

DOUBLE COMPLEX for zsytri_rook.

Arrays:

a(size *lda* by *) contains the factorization of the matrix *A*, as returned by ?sytrf_rook.

The second dimension of *a* must be at least max(1,*n*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,*n*).

lda — INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).

ipiv — INTEGER.

Array, size at least max(1, *n*).

The *ipiv* array, as returned by ?sytrf_rook.

Output Parameters

a — Overwritten by the *n*-by-*n* matrix inv(*A*). If *uplo* = 'U', the upper triangular part of the inverse is formed and the part of *a* below the diagonal is not referenced; if *uplo* = 'L' the lower triangular part of the inverse is formed and the part of *a* above the diagonal is not referenced."

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* =-*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sytri_rook interface are as follows:

a — Holds the matrix *A* of size (*n*,*n*).

ipiv — Holds the vector of length *n*.

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?hetri
*Computes the inverse of a complex Hermitian matrix using U*D*U^H or L*D*L^H Bunch-Kaufman factorization.*

Syntax

```
call chetri( uplo, n, a, lda, ipiv, work, info )
call zhetri( uplo, n, a, lda, ipiv, work, info )
call hetri( a, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a complex Hermitian matrix *A*. Before calling this routine, call ?hetrf to factorize *A*.

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the array *a* stores the Bunch-Kaufman factorization $A = U*D*U^H$.
If *uplo* = 'L', the array *a* stores the Bunch-Kaufman factorization $A = L*D*L^H$.

n
INTEGER. The order of the matrix *A*; $n \geq 0$.

a, *work*
COMPLEX for chetri
DOUBLE COMPLEX for zhetri.
Arrays:
a(*lda*,*) contains the factorization of the matrix *A*, as returned by ?hetrf.
The second dimension of *a* must be at least max(1,*n*).
work(*) is a workspace array.
The dimension of *work* must be at least max(1,*n*).

lda
INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).

ipiv
INTEGER.
Array, size at least max(1, *n*). The *ipiv* array, as returned by ?hetrf.

Output Parameters

a
Overwritten by the *n*-by-*n* matrix inv(*A*).

info
INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hetri` interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^H*P^T*X*P*U - I| \le c(n)\varepsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = `'U'`, and

$$|D*L^H*P^T*X*P*L - I| \le c(n)\varepsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = `'L'`. Here $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is `?sytri`.

See Also

Matrix Storage Schemes

?hetri_rook
*Computes the inverse of a complex Hermitian matrix using U*D*U^H or L*D*L^H bounded Bunch-Kaufman factorization.*

Syntax

```
call chetri_rook( uplo, n, a, lda, ipiv, work, info )
call zhetri_rook( uplo, n, a, lda, ipiv, work, info )
call hetri_rook( a, ipiv [,uplo] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes the inverse `inv(A)` of a complex Hermitian matrix *A*. Before calling this routine, call `?hetrf_rook` to factorize *A*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the array *a* stores the factorization $A = U*D*U^H$. If *uplo* = 'L', the array *a* stores the factorization $A = L*D*L^H$.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
a, *work*	COMPLEX for chetri_rook DOUBLE COMPLEX for zhetri_rook. Arrays: *a*(*lda*,*) contains the factorization of the matrix *A*, as returned by ?hetrf_rook. The second dimension of *a* must be at least max(1,*n*). *work*(*) is a workspace array. The dimension of *work* must be at least max(1,*n*).
lda	INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).
ipiv	INTEGER. Array, size at least max(1, *n*). The *ipiv* array, as returned by ?hetrf_rook.

Output Parameters

a	Overwritten by the *n*-by-*n* matrix inv(*A*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hetri_rook interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
ipiv	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations is approximately $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is ?sytri_rook.

See Also

Matrix Storage Schemes

?sytri2

Computes the inverse of a symmetric indefinite matrix through setting the leading dimension of the workspace and calling ?sytri2x*.*

Syntax

```
call ssytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call csytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call sytri2( a,ipiv[,uplo][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a symmetric indefinite matrix *A* using the factorization $A = U*D*U^T$ or $A = L*D*L^T$ computed by ?sytrf.

The ?sytri2 routine sets the leading dimension of the workspace before calling ?sytri2x that actually computes the inverse.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the array *a* stores the factorization $A = U*D*U^T$. If *uplo* = 'L', the array *a* stores the factorization $A = L*D*L^T$.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
a, *work*	REAL for ssytri2 DOUBLE PRECISION for dsytri2 COMPLEX for csytri2 DOUBLE COMPLEX for zsytri2 Array *a*(size *lda* by *n*) contains the block diagonal matrix *D* and the multipliers used to obtain the factor U or L as returned by ?sytrf. The second dimension of *a* must be at least max(1,*n*). *work* is a workspace array of (*n*+*nb*+1)*(*nb*+3) dimension.
lda	INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.
ipiv	INTEGER.

Array, size at least max(1, *n*).

Details of the interchanges and the block structure of *D* as returned by ?sytrf.

lwork INTEGER. The dimension of the *work* array.

lwork≥ (*n*+*nb*+1)*(*nb*+3)

where

nb is the block size parameter as returned by sytrf.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

a If *info* = 0, the symmetric inverse of the original matrix.

If *uplo* = 'U', the upper triangular part of the inverse is formed and the part of *A* below the diagonal is not referenced.

If *uplo* = 'L', the lower triangular part of the inverse is formed and the part of *A* above the diagonal is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* =-*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, *D*(*i*,*i*) = 0; *D* is singular and its inversion could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sytri2 interface are as follows:

a Holds the matrix *A* of size (*n*,*n*).

ipiv Holds the vector of length *n*.

uplo Indicates how the matrix *A* has been factored. Must be 'U' or 'L'.

See Also

?sytrf

?sytri2x

Matrix Storage Schemes

?hetri2
Computes the inverse of a Hermitian indefinite matrix through setting the leading dimension of the workspace and calling ?hetri2x*.*

Syntax

```
call chetri2( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetri2( uplo, n, a, lda, ipiv, work, lwork, info )
call hetri2( a,ipiv[,uplo][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a Hermitian indefinite matrix *A* using the factorization $A = U*D*U^H$ or $A = L*D*L^H$ computed by ?hetrf.

The ?hetri2 routine sets the leading dimension of the workspace before calling ?hetri2x that actually computes the inverse.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *a* stores the factorization $A = U*D*U^H$.

If *uplo* = 'L', the array *a* stores the factorization $A = L*D*L^H$.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

a, *work*

COMPLEX for chetri2

DOUBLE COMPLEX for zhetri2

Array *a*(size *lda* by *) contains the block diagonal matrix *D* and the multipliers used to obtain the factor U or L as returned by ?sytrf.

The second dimension of *a* must be at least max(1,*n*).

work is a workspace array of (*n*+*nb*+1)*(*nb*+3) dimension.

lda

INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ipiv

INTEGER.

Array, size at least max(1, *n*).

Details of the interchanges and the block structure of *D* as returned by ?hetrf.

lwork

INTEGER. The dimension of the *work* array.

$lwork \geq (n+nb+1)*(nb+3)$

where

nb is the block size parameter as returned by hetrf.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

a	If *info* = 0, the inverse of the original matrix. If *uplo* = 'U', the upper triangular part of the inverse is formed and the part of *A* below the diagonal is not referenced. If *uplo* = 'L', the lower triangular part of the inverse is formed and the part of *A* above the diagonal is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* =-*i*, the *i*-th parameter had an illegal value. If *info* = *i*, *D*(*i*,*i*) = 0; *D* is singular and its inversion could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hetri2 interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
ipiv	Holds the vector of length *n*.
uplo	Indicates how the input matrix *A* has been factored. Must be 'U' or 'L'.

See Also

?hetrf

?hetri2x

Matrix Storage Schemes

?sytri2x

Computes the inverse of a symmetric indefinite matrix after ?sytri2*sets the leading dimension of the workspace.*

Syntax

```
call ssytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call dsytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call csytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call zsytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call sytri2x( a,ipiv,nb[,uplo][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse `inv(A)` of a symmetric indefinite matrix *A* using the factorization $A = U*D*U^T$ or $A = L*D*L^T$ computed by `?sytrf`.

The `?sytri2x` actually computes the inverse after the `?sytri2` routine sets the leading dimension of the workspace before calling `?sytri2x`.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates how the input matrix *A* has been factored:

If *uplo* = `'U'`, the array *a* stores the factorization $A = U*D*U^T$.

If *uplo* = `'L'`, the array *a* stores the factorization $A = L*D*L^T$.

n

`INTEGER`. The order of the matrix *A*; $n \geq 0$.

a, *work*

`REAL` for `ssytri2x`

`DOUBLE PRECISION` for `dsytri2x`

`COMPLEX` for `csytri2x`

`DOUBLE COMPLEX` for `zsytri2x`

Arrays:

a(*lda*,*) contains the *nb* (block size) diagonal matrix *D* and the multipliers used to obtain the factor `U` or `L` as returned by `?sytrf`.

The second dimension of *a* must be at least `max(1,n)`.

work is a workspace array of dimension `(n+nb+1)*(nb+3)`

where

nb is the block size as set by `?sytrf`.

lda

`INTEGER`. The leading dimension of *a*; $lda \geq \max(1, n)$.

ipiv

`INTEGER`.

Array, size at least `max(1, n)`.

Details of the interchanges and the *nb* structure of *D* as returned by `?sytrf`.

nb

`INTEGER`. Block size.

Output Parameters

a

If *info* = 0, the symmetric inverse of the original matrix.

If *info* = `'U'`, the upper triangular part of the inverse is formed and the part of *A* below the diagonal is not referenced.

If *info* = `'L'`, the lower triangular part of the inverse is formed and the part of *A* above the diagonal is not referenced.

info

`INTEGER`.

If *info* = 0, the execution is successful.

If *info* =-*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, D_{ii}= 0; *D* is singular and its inversion could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sytri2x` interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
ipiv	Holds the vector of length *n*.
nb	Holds the block size.
uplo	Indicates how the input matrix *A* has been factored. Must be `'U'` or `'L'`.

See Also

?sytrf

?sytri2

Matrix Storage Schemes

?hetri2x
Computes the inverse of a Hermitian indefinite matrix after `?hetri2`*sets the leading dimension of the workspace.*

Syntax

```
call chetri2x( uplo, n, a, lda, ipiv, work, nb, info )
call zhetri2x( uplo, n, a, lda, ipiv, work, nb, info )
call hetri2x( a,ipiv,nb[,uplo][,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes the inverse `inv(A)` of a Hermitian indefinite matrix *A* using the factorization $A = U*D*U^H$ or $A = L*D*L^H$ computed by `?hetrf`.

The `?hetri2x` actually computes the inverse after the `?hetri2` routine sets the leading dimension of the workspace before calling `?hetri2x`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates how the input matrix *A* has been factored: If *uplo* = `'U'`, the array *a* stores the factorization $A = U*D*U^H$. If *uplo* = `'L'`, the array *a* stores the factorization $A = L*D*L^H$.

n	INTEGER. The order of the matrix *A*; *n* ≥ 0.
a, *work*	COMPLEX for chetri2x DOUBLE COMPLEX for zhetri2x Arrays: *a*(*lda*,*) contains the *nb* (block size) diagonal matrix *D* and the multipliers used to obtain the factor U or L as returned by ?hetrf. The second dimension of *a* must be at least max(1,*n*). *work* is a workspace array of the dimension (*n*+*nb*+1)*(*nb*+3) where *nb* is the block size as set by ?hetrf.
lda	INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).
ipiv	INTEGER. Array, size at least max(1, *n*). Details of the interchanges and the *nb* structure of *D* as returned by ?hetrf.
nb	INTEGER. Block size.

Output Parameters

a	If *info* = 0, the symmetric inverse of the original matrix. If *info* = 'U', the upper triangular part of the inverse is formed and the part of *A* below the diagonal is not referenced. If *info* = 'L', the lower triangular part of the inverse is formed and the part of *A* above the diagonal is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* =-*i*, the *i*-th parameter had an illegal value. If *info* = *i*, D_{ii} = 0; *D* is singular and its inversion could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hetri2x interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
ipiv	Holds the vector of length *n*.
nb	Holds the block size.

uplo Indicates how the input matrix *A* has been factored. Must be `'U'` or `'L'`.

See Also

`?hetrf`

`?hetri2`

Matrix Storage Schemes

?sytri_3
Computes the inverse of a real or complex symmetric matrix.

```
call ssytri_3(uplo, n, A, lda, e, ipiv, work, lwork, info)
call dsytri_3(uplo, n, A, lda, e, ipiv, work, lwork, info)
call csytri_3(uplo, n, A, lda, e, ipiv, work, lwork, info)
call zsytri_3(uplo, n, A, lda, e, ipiv, work, lwork, info)
```

Description

`?sytri_3` computes the inverse of a real or complex symmetric matrix A using the factorization computed by `?sytrf_rk`: A = P*U*D*(U^T)*(P^T) or A = P*L*D*(L^T)*(P^T), where U (or L) is a unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

`?sytri_3` sets the leading dimension of the workspace before calling `?sytri_3x`, which actually computes the inverse. This is the blocked version of the algorithm, calling Level-3 BLAS.

Input Parameters

uplo `CHARACTER*1`

Specifies whether the details of the factorization are stored as an upper or lower triangular matrix.

- = `'U'`: The upper triangle of A is stored.
- = `'L'`: The lower triangle of A is stored.

n `INTEGER`

The order of the matrix A. $n \geq 0$.

A `REAL` for `ssytri_3`

`DOUBLE PRECISION` for `dsytri_3`

`COMPLEX` for `csytri_3`

`COMPLEX*16` for `zsytri_3`

Array, dimension (*lda*,*n*). On entry, diagonal of the block diagonal matrix D and factors U or L as computed by `?sytrf_rk`:

- *Only* diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, D(*k*,*k*) = A(*k*,*k*). Superdiagonal (or subdiagonal) elements of D should be provided on entry in array *e*.

  —and—

- If *uplo* = 'U', factor U in the superdiagonal part of A. If *uplo* = 'L', factor L in the subdiagonal part of A.

lda INTEGER

The leading dimension of the array *A*.*lda* ≥ max(1, *n*).

e REAL for ssytri_3

DOUBLE PRECISION for dsytri_3

COMPLEX for csytri_3

COMPLEX*16 for zsytri_3

Array, dimension (*n*). On entry, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', e(*i*) = D(*i*–1,*i*), *i*=2:N, and e(1) is not referenced. If *uplo* = 'L', e(*i*) = D(*i*+1,*i*), *i*=1:N–1, and e(*n*) is not referenced.

NOTE For 1-by-1 diagonal block D(*k*), where 1 ≤ k ≤ *n*, the element *e*(*k*) is not referenced in both the *uplo* = 'U' and *uplo* = 'L' cases.

ipiv INTEGER

Array, dimension (*n*). Details of the interchanges and the block structure of D as determined by ?sytrf_rk.

lwork INTEGER

The length of the array *work*.

If *LDWORK* = *-1*, a workspace query is assumed; the routine calculates only the optimal size of the optimal size of the *work* array and returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by XERBLA.

Output Parameters

A REAL for ssytri_3

DOUBLE PRECISION for dsytri_3

COMPLEX for csytri_3

COMPLEX*16 for zsytri_3

On exit, if *info* = 0, the symmetric inverse of the original matrix. If *uplo* = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If *uplo* = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

work REAL for ssytri_3

DOUBLE PRECISION for dsytri_3

COMPLEX for csytri_3

COMPLEX*16 for zsytri_3

Array, dimension (*n*+*NB*+1)*(*NB*+3). On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info INTEGER

- = 0: successful exit.
- < 0: If *info* = –*i*, the *i*th argument had an illegal value.
- > 0: If *info* = *i*, D(*i*,*i*) = 0; the matrix is singular and its inverse could not be computed.

?hetri_3

Computes the inverse of a complex Hermitian matrix using the factorization computed by ?hetrf_rk*.*

```
call chetri_3(uplo, n, A, lda, e, ipiv, work, lwork, info)
call zhetri_3(uplo, n, A, lda, e, ipiv, work, lwork, info)
```

Description

?hetri_3 computes the inverse of a complex Hermitian matrix A using the factorization computed by ?hetrf_rk: A = P*U*D*(U^H)*(P^T) or A = P*L*D*(L^H)*(P^T), where U (or L) is a unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?hetri_3 sets the leading dimension of the workspace before calling ?hetri_3x, which actually computes the inverse.

This is the blocked version of the algorithm, calling Level-3 BLAS.

Input Parameters

uplo CHARACTER*1

Specifies whether the details of the factorization are stored as an upper or lower triangular matrix.

- = 'U': The upper triangle of A is stored.
- = 'L': The lower triangle of A is stored.

n INTEGER

The order of the matrix A. *n* ≥ 0.

A COMPLEX for chetri_3

COMPLEX*16 for zhetri_3

Array, dimension (*lda*,*n*). On entry, diagonal of the block diagonal matrix D and factor U or L as computed by ?hetrf_rk:

- *Only* diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, D(*k*,*k*) = A(*k*,*k*). Superdiagonal (or subdiagonal) elements of D should be provided on entry in array *e*.
- If *uplo* = 'U', factor U in the superdiagonal part of A. If *uplo* = 'L', factor L is the subdiagonal part of A.

lda INTEGER

The leading dimension of the array *A*. *lda* ≥ max(1, *n*).

e

COMPLEX for chetri_3

COMPLEX*16 for zhetri_3

Array, dimension (*n*). On entry, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', e(*i*) = D(*i*–1,*i*), *i*=2:N, and e(1) is not referenced. If *uplo* = 'L', e(*i*) = D(*i*+1,*i*), *i*=1:N–1, and e(*n*) is not referenced.

NOTE For 1-by-1 diagonal block D(*k*), where 1 ≤ *k* ≤ *n*, the element *e*(*k*) is not referenced in both the *uplo* = 'U' and *uplo* = 'L' cases.

ipiv

INTEGER

Array, dimension (*n*). Details of the interchanges and the block structure of D as determined by ?hetrf_rk.

lwork

INTEGER

The length of the array *work*.

If *LDWORK* = –1, a workspace query is assumed; the routine calculates only the optimal size of the *work* array and returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by XERBLA.

Output Parameters

A

COMPLEX for chetri_3

COMPLEX*16 for zhetri_3

On exit, if *info* = 0, the Hermitian inverse of the original matrix. If *uplo* = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If *uplo* = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

work

COMPLEX for chetri_3

COMPLEX*16 for zhetri_3

Array, dimension (*n*+*NB*+1)*(*NB*+3). On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info

INTEGER

- = 0: Successful exit.
- < 0: If *info* = *-i*, the i^{th} argument had an illegal value.
- > 0: If *info* = *i*, D(*i*,*i*) = 0; the matrix is singular and its inverse could not be computed.

?sptri

Computes the inverse of a symmetric matrix using $U*D*U^T$ or $L*D*L^T$ Bunch-Kaufman factorization of matrix in packed storage.

Syntax

```
call ssptri( uplo, n, ap, ipiv, work, info )
call dsptri( uplo, n, ap, ipiv, work, info )
call csptri( uplo, n, ap, ipiv, work, info )
call zsptri( uplo, n, ap, ipiv, work, info )
call sptri( ap, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a packed symmetric matrix *A*. Before calling this routine, call ?sptrf to factorize *A*.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *ap* stores the Bunch-Kaufman factorization $A = U*D*U^T$.

If *uplo* = 'L', the array *ap* stores the Bunch-Kaufman factorization $A = L*D*L^T$.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

ap, *work*

REAL for ssptri

DOUBLE PRECISION for dsptri

COMPLEX for csptri

DOUBLE COMPLEX for zsptri.

Arrays:

ap(*) contains the factorization of the matrix *A*, as returned by ?sptrf.

The dimension of *ap* must be at least max(1,*n*(*n*+1)/2).

work(*) is a workspace array.

The dimension of *work* must be at least max(1,*n*).

ipiv

INTEGER.

Array, size at least max(1, *n*). The *ipiv* array, as returned by ?sptrf.

Output Parameters

ap	Overwritten by the matrix `inv(A)` in packed form.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sptri` interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^T*P^T*X*P*U - I| \le c(n)\varepsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = `'U'`, and

$$|D*L^T*P^T*X*P*L - I| \le c(n)\varepsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = `'L'`. Here $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?hptri
*Computes the inverse of a complex Hermitian matrix using U*D*U^H or L*D*L^H Bunch-Kaufman factorization of matrix in packed storage.*

Syntax

```
call chptri( uplo, n, ap, ipiv, work, info )
call zhptri( uplo, n, ap, ipiv, work, info )
call hptri( ap, ipiv [,uplo] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes the inverse `inv(A)` of a complex Hermitian matrix *A* using packed storage. Before calling this routine, call `?hptrf` to factorize *A*.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates how the input matrix *A* has been factored:

If *uplo* = `'U'`, the array *ap* stores the packed Bunch-Kaufman factorization $A = U*D*U^H$.

If *uplo* = `'L'`, the array *ap* stores the packed Bunch-Kaufman factorization $A = L*D*L^H$.

n

`INTEGER`. The order of the matrix *A*; $n \geq 0$.

ap, *work*

`COMPLEX` for `chptri`

`DOUBLE COMPLEX` for `zhptri`.

Arrays:

ap`(*)` contains the factorization of the matrix *A*, as returned by `?hptrf`.

The dimension of *ap* must be at least `max(1,`*n*`(`*n*`+1)/2)`.

work`(*)` is a workspace array.

The dimension of *work* must be at least `max(1,`*n*`)`.

ipiv

`INTEGER`.

Array, size at least `max(1,` *n*`)`.

The *ipiv* array, as returned by `?hptrf`.

Output Parameters

ap

Overwritten by the matrix `inv(`*A*`)`.

info

`INTEGER`.

If *info* = `0`, the execution is successful.

If *info* = `-`*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hptri` interface are as follows:

ap

Holds the array *A* of size (*n*`*(`*n*`+1)/2`).

ipiv	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^H*P^T*X*P*U - I| \le c(n)\varepsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D*L^H*P^T*X*PL - I| \le c(n)\varepsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(8/3)n^3$.

The real counterpart of this routine is ?sptri.

See Also

Matrix Storage Schemes

?trtri
Computes the inverse of a triangular matrix.

Syntax

```
call strtri( uplo, diag, n, a, lda, info )
call dtrtri( uplo, diag, n, a, lda, info )
call ctrtri( uplo, diag, n, a, lda, info )
call ztrtri( uplo, diag, n, a, lda, info )
call trtri( a [,uplo] [,diag] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a triangular matrix *A*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether *A* is upper or lower triangular: If *uplo* = 'U', then *A* is upper triangular. If *uplo* = 'L', then *A* is lower triangular.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.

n	INTEGER. The order of the matrix *A*; *n* ≥ 0.
a	REAL for strtri DOUBLE PRECISION for dtrtri COMPLEX for ctrtri DOUBLE COMPLEX for ztrtri. Array: size *lda* by *size max(1,*lda**n*). Contains the matrix *A*. The second dimension of *a* must be at least max(1,*n*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

Output Parameters

a	Overwritten by the matrix inv(*A*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *A* is zero, *A* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine trtri interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|XA - I| \le c(n)\varepsilon\ |X||A|$$

$$|XA - I| \le c(n)\varepsilon\ |A^{-1}||A||X|,$$

where $c(n)$ is a modest linear function of *n*; ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?tftri
Computes the inverse of a triangular matrix stored in the Rectangular Full Packed (RFP) format.

Syntax

```
call stftri( transr, uplo, diag, n, a, info )
call dtftri( transr, uplo, diag, n, a, info )
call ctftri( transr, uplo, diag, n, a, info )
call ztftri( transr, uplo, diag, n, a, info )
```

Include Files

- mkl.fi, lapack.f90

Description

Computes the inverse of a triangular matrix *A* stored in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

This is the block version of the algorithm, calling Level 3 BLAS.

Input Parameters

transr	CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data). If *transr* = 'N', the Normal *transr* of RFP *A* is stored. If *transr* = 'T', the Transpose *transr* of RFP *A* is stored. If *transr* = 'C', the Conjugate-Transpose *transr* of RFP *A* is stored.
uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of RFP *A* is stored: If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*. If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
a	REAL for stftri DOUBLE PRECISION for dtftri COMPLEX for ctftri DOUBLE COMPLEX for ztftri. Array, size max(1, *n**(*n* + 1)/2). The array *a* contains the matrix *A* in the RFP format.

Output Parameters

a	The (triangular) inverse of the original matrix in the same storage format.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, *A*(*i*,*i*) is exactly zero. The triangular matrix is singular and its inverse cannot be computed.

See Also

Matrix Storage Schemes

?tptri
Computes the inverse of a triangular matrix using packed storage.

Syntax

```
call stptri( uplo, diag, n, ap, info )
call dtptri( uplo, diag, n, ap, info )
call ctptri( uplo, diag, n, ap, info )
call ztptri( uplo, diag, n, ap, info )
call tptri( ap [,uplo] [,diag] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse inv(*A*) of a packed triangular matrix *A*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether *A* is upper or lower triangular: If *uplo* = 'U', then *A* is upper triangular. If *uplo* = 'L', then *A* is lower triangular.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *ap*.
n	INTEGER. The order of the matrix *A*; *n*≥ 0.
ap	REAL for stptri DOUBLE PRECISION for dtptri

`COMPLEX` for `ctptri`

`DOUBLE COMPLEX` for `ztptri`.

Array, size at least `max(1,n(n+1)/2)`.

Contains the packed triangular matrix A.

Output Parameters

ap — Overwritten by the packed *n*-by-*n* matrix `inv(A)` .

info — `INTEGER`.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is zero, *A* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `tptri` interface are as follows:

ap — Holds the array *A* of size (*n**(*n*+1)/2).

uplo — Must be `'U'` or `'L'`. The default value is `'U'`.

diag — Must be `'N'` or `'U'`. The default value is `'N'`.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|XA - I| \le c(n)\varepsilon\ |X||A|$$

$$|X - A^{-1}| \le c(n)\varepsilon\ |A^{-1}||A||X|,$$

where $c(n)$ is a modest linear function of *n*; ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

Matrix Equilibration: LAPACK Computational Routines

Routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

?geequ
Computes row and column scaling factors intended to equilibrate a general matrix and reduce its condition number.

Syntax

```
call sgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call dgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call cgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call zgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call geequ( a, r, c [,rowcnd] [,colcnd] [,amax] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate an *m*-by-*n* matrix *A* and reduce its condition number. The output array *r* returns the row scale factors and the array *c* the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix *B* with elements b_{ij}=r(i)*a_{ij}*c(j) have absolute value 1.

See ?laqge auxiliary function that uses scaling factors computed by ?geequ.

Input Parameters

m

INTEGER. The number of rows of the matrix *A*; *m*≥ 0.

n

INTEGER. The number of columns of the matrix *A*; *n*≥ 0.

a

REAL for sgeequ

DOUBLE PRECISION for dgeequ

COMPLEX for cgeequ

DOUBLE COMPLEX for zgeequ.

Array: size *lda* by *.

Contains the *m*-by-*n* matrix *A* whose equilibration factors are to be computed.

The second dimension of *a* must be at least max(1,*n*).

lda

INTEGER. The leading dimension of *a*; *lda*≥ max(1, *m*).

Output Parameters

r, *c*

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r* (size *m*), *c* (size *n*).

If *info* = 0, or *info*>*m*, the array *r* contains the row scale factors of the matrix *A*.

If *info* = 0, the array *c* contains the column scale factors of the matrix *A*.

rowcnd

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If *info* = 0 or *info*>*m*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i).

colcnd — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If *info* = 0, *colcnd* contains the ratio of the smallest *c*(i) to the largest *c*(i).

amax — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix *A*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, *i* > 0, and

i≤*m*, the *i*-th row of *A* is exactly zero;

i>*m*, the (*i*-*m*)th column of *A* is exactly zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine geequ interface are as follows:

a — Holds the matrix *A* of size (*m*, *n*).

r — Holds the vector of length (*m*).

c — Holds the vector of length *n*.

Application Notes

All the components of *r* and *c* are restricted to be between SMLNUM = smallest safe number and BIGNUM= largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *A* but works well in practice.

SMLNUM and *BIGNUM* are parameters representing machine precision. You can use the ?lamch routines to compute them. For example, compute single precision values of *SMLNUM* and *BIGNUM* as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

If *rowcnd*≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r*.

If *colcnd*≥ 0.1, it is not worth scaling by *c*.

If *amax* is very close to SMLNUM or very close to BIGNUM, the matrix *A* should be scaled.

See Also

Error Analysis

Matrix Storage Schemes

?geequb
Computes row and column scaling factors restricted to a power of radix to equilibrate a general matrix and reduce its condition number.

Syntax

```
call sgeequb( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call dgeequb( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call cgeequb( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call zgeequb( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate an *m*-by-*n* general matrix *A* and reduce its condition number. The output array *r* returns the row scale factors and the array *c* - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix *B* with elements $b_{i,j}$ =r(i)*$a_{i,j}$*c(j) have an absolute value of at most the radix.

r(i) and *c*(j) are restricted to be a power of the radix between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *a* but works well in practice.

SMLNUM and *BIGNUM* are parameters representing machine precision. You can use the ?lamch routines to compute them. For example, compute single precision values of *SMLNUM* and *BIGNUM* as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from ?geequ by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between sqrt(radix) and 1/sqrt(radix).

Input Parameters

m
INTEGER. The number of rows of the matrix *A*; *m*≥ 0.

n
INTEGER. The number of columns of the matrix *A*; *n*≥ 0.

a
REAL for sgeequb

DOUBLE PRECISION for dgeequb

COMPLEX for cgeequb

DOUBLE COMPLEX for zgeequb.

Array: size (*lda*,*).

Contains the *m*-by-*n* matrix *A* whose equilibration factors are to be computed.

The second dimension of *a* must be at least max(1,*n*).

lda	INTEGER. The leading dimension of *a*; *lda*≥ max(1, *m*).

Output Parameters

r, *c*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: *r*(*m*), *c*(*n*). If *info* = 0, or *info*>*m*, the array *r* contains the row scale factors for the matrix *A*. If *info* = 0, the array *c* contains the column scale factors for the matrix *A*.
rowcnd	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *info* = 0 or *info*>*m*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i). If *rowcnd*≥ 0.1, and *amax* is neither too large nor too small, it is not worth scaling by *r*.
colcnd	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *info* = 0, *colcnd* contains the ratio of the smallest *c*(i) to the largest *c*(i). If *colcnd*≥ 0.1, it is not worth scaling by *c*.
amax	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix *A*. If *amax* is very close to SMLNUM or very close to BIGNUM, the matrix should be scaled.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, *i* > 0, and *i*≤*m*, the *i*-th row of *A* is exactly zero; *i*>*m*, the (*i*-*m*)-th column of *A* is exactly zero.

See Also

Error Analysis
Matrix Storage Schemes

?gbequ

Computes row and column scaling factors intended to equilibrate a banded matrix and reduce its condition number.

Syntax

```
call sgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

```
call dgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call cgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call zgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call gbequ( ab, r, c [,kl] [,rowcnd] [,colcnd] [,amax] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate an *m*-by-*n* band matrix *A* and reduce its condition number. The output array *r* returns the row scale factors and the array *c* the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix *B* with elements b_{ij}=r(i)*a_{ij}*c(j) have absolute value 1.

See ?laqgb auxiliary function that uses scaling factors computed by ?gbequ.

Input Parameters

m
: INTEGER. The number of rows of the matrix *A*; *m*≥ 0.

n
: INTEGER. The number of columns of the matrix *A*; *n*≥ 0.

kl
: INTEGER. The number of subdiagonals within the band of *A*; *kl*≥ 0.

ku
: INTEGER. The number of superdiagonals within the band of *A*; *ku*≥ 0.

ab
: REAL for sgbequ

DOUBLE PRECISION for dgbequ

COMPLEX for cgbequ

DOUBLE COMPLEX for zgbequ.

Array, size *ldab* by *. Contains the original band matrix *A* stored in rows from 1 to *kl* + *ku* + 1. The second dimension of *ab* must be at least max(1,*n*).

ldab
: INTEGER. The leading dimension of *ab*; *ldab*≥*kl*+*ku*+1.

Output Parameters

r, *c*
: REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r* (size *m*), *c* (size *n*).

If *info* = 0, or *info*>*m*, the array *r* contains the row scale factors of the matrix *A*.

If *info* = 0, the array *c* contains the column scale factors of the matrix *A*.

rowcnd
: REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

	If *info* = 0 or *info*>*m*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i).
colcnd	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *info* = 0, *colcnd* contains the ratio of the smallest *c*(i) to the largest *c*(i).
amax	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix *A*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i* and *i*≤*m*, the *i*-th row of *A* is exactly zero; *i*>*m*, the (*i*-*m*)th column of *A* is exactly zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gbequ interface are as follows:

ab	Holds the array *A* of size (*kl*+*ku*+1, *n*).
r	Holds the vector of length (*m*).
c	Holds the vector of length *n*.
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *ku* = *lda*-*kl*-1.

Application Notes

All the components of *r* and *c* are restricted to be between *SMLNUM* = smallest safe number and *BIGNUM*= largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *A* but works well in practice.

SMLNUM and *BIGNUM* are parameters representing machine precision. You can use the ?lamch routines to compute them. For example, compute single precision values of *SMLNUM* and *BIGNUM* as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

If *rowcnd*≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r*.

If *colcnd*≥ 0.1, it is not worth scaling by *c*.

If *amax* is very close to SMLNUM or very close to BIGNUM, the matrix *A* should be scaled.

See Also

Error Analysis
Matrix Storage Schemes

?gbequb
Computes row and column scaling factors restricted to a power of radix to equilibrate a banded matrix and reduce its condition number.

Syntax

```
call sgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call dgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call cgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call zgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate an *m*-by-*n* banded matrix *A* and reduce its condition number. The output array *r* returns the row scale factors and the array *c* - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix *B* with elements *b(ij)=r(i)*a(ij)*c(j)* have an absolute value of at most the radix.

r(i) and *c(j)* are restricted to be a power of the radix between *SMLNUM* = smallest safe number and *BIGNUM* = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *a* but works well in practice.

SMLNUM and *BIGNUM* are parameters representing machine precision. You can use the ?lamch routines to compute them. For example, compute single precision values of *SMLNUM* and *BIGNUM* as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from ?gbequ by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between sqrt(radix) and 1/sqrt(radix).

Input Parameters

m	INTEGER. The number of rows of the matrix *A*; *m*≥ 0.
n	INTEGER. The number of columns of the matrix *A*; *n*≥ 0.
kl	INTEGER. The number of subdiagonals within the band of *A*; *kl*≥ 0.
ku	INTEGER. The number of superdiagonals within the band of *A*; *ku*≥ 0.
ab	REAL for sgbequb DOUBLE PRECISION for dgbequb COMPLEX for cgbequb DOUBLE COMPLEX for zgbequb.

Array: size *ldab* by *

Contains the original banded matrix *A* stored in rows from 1 to *kl* + *ku* + 1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

ab(*ku*+1+*i*-*j*,*j*) = *a*(*i*,*j*) for max(1,*j*-*ku*) ≤*i*≤ min(*n*,*j*+*kl*).

The second dimension of *ab* must be at least max(1,*n*).

ldab INTEGER. The leading dimension of *a*; *ldab*≥ max(1, *m*).

Output Parameters

r, *c* REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r* (size *m*), *c* (size *n*).

If *info* = 0, or *info*>*m*, the array *r* contains the row scale factors for the matrix *A*.

If *info* = 0, the array *c* contains the column scale factors for the matrix *A*.

rowcnd REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If *info* = 0 or *info*>*m*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i). If *rowcnd*≥ 0.1, and *amax* is neither too large nor too small, it is not worth scaling by *r*.

colcnd REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If *info* = 0, *colcnd* contains the ratio of the smallest *c*(i) to the largest *c*(i). If *colcnd*≥ 0.1, it is not worth scaling by *c*.

amax REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix *A*. If *amax* is very close to SMLNUM or BIGNUM, the matrix should be scaled.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

i≤*m*, the *i*-th row of *A* is exactly zero;

i>*m*, the (*i*-*m*)-th column of *A* is exactly zero.

See Also

Error Analysis
Matrix Storage Schemes

?poequ
Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

```
call spoequ( n, a, lda, s, scond, amax, info )
call dpoequ( n, a, lda, s, scond, amax, info )
call cpoequ( n, a, lda, s, scond, amax, info )
call zpoequ( n, a, lda, s, scond, amax, info )
call poequ( a, s [,scond] [,amax] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix *A* and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors such that `s(i)s[i + 1]` contains

$$1/\sqrt{a_{i,i}}$$

These factors are chosen so that the scaled matrix *B* with elements $B_{i,j}=s(i)*A_{i,j}*s(j)$ has diagonal elements equal to 1.

This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

See `?laqsy` auxiliary function that uses scaling factors computed by `?poequ`.

Input Parameters

n	`INTEGER`. The order of the matrix *A*; $n \geq 0$.

a	`REAL` for `spoequ`
	`DOUBLE PRECISION` for `dpoequ`
	`COMPLEX` for `cpoequ`
	`DOUBLE COMPLEX` for `zpoequ`.
	Array: size *lda* by *.
	Contains the *n*-by-*n* symmetric or Hermitian positive definite matrix *A* whose scaling factors are to be computed. Only the diagonal elements of *A* are referenced.
	The second dimension of *a* must be at least `max(1,n)`.

lda	`INTEGER`. The leading dimension of *a*; $lda \geq$ `max(1,n)`.

Output Parameters

s	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size *n*. If *info* = 0, the array *s* contains the scale factors for *A*.
scond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i).
amax	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix *A*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine poequ interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
s	Holds the vector of length *n*.

Application Notes

If *scond*≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to SMLNUM or very close to BIGNUM, the matrix *A* should be scaled.

See Also

Error Analysis
Matrix Storage Schemes

?poequb

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

```
call spoequb( n, a, lda, s, scond, amax, info )
call dpoequb( n, a, lda, s, scond, amax, info )
call cpoequb( n, a, lda, s, scond, amax, info )
```

```
call zpoequb( n, a, lda, s, scond, amax, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix *A* and reduce its condition number (with respect to the two-norm).

These factors are chosen so that the scaled matrix *B* with elements $B_{i,j}$=s(i)*$A_{i,j}$*s(j) has diagonal elements equal to 1. *s(i)* is a power of two nearest to, but not exceeding 1/sqrt($A_{i,i}$).

This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

n

INTEGER. The order of the matrix *A*; *n*≥ 0.

a

REAL for spoequb

DOUBLE PRECISION for dpoequb

COMPLEX for cpoequb

DOUBLE COMPLEX for zpoequb.

Array: size *lda* by *.

Contains the *n*-by-*n* symmetric or Hermitian positive definite matrix *A* whose scaling factors are to be computed. Only the diagonal elements of *A* are referenced.

The second dimension of *a* must be at least max(1,*n*).

lda

INTEGER. The leading dimension of *a*; *lda*≥ max(1, *m*).

Output Parameters

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*).

If *info* = 0, the array *s* contains the scale factors for *A*.

scond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i). If *scond*≥ 0.1, and *amax* is neither too large nor too small, it is not worth scaling by *s*.

amax

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

	Absolute value of the largest element of the matrix *A*. If *amax* is very close to SMLNUM or BIGNUM, the matrix should be scaled.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

See Also

Error Analysis
Matrix Storage Schemes

?ppequ
Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix in packed storage and reduce its condition number.

Syntax

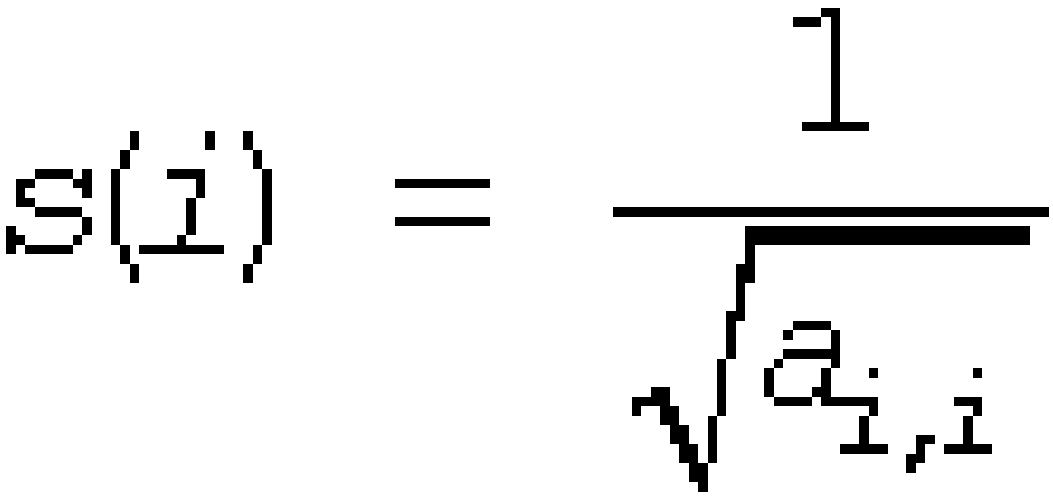
```
call sppequ( uplo, n, ap, s, scond, amax, info )
call dppequ( uplo, n, ap, s, scond, amax, info )
call cppequ( uplo, n, ap, s, scond, amax, info )
call zppequ( uplo, n, ap, s, scond, amax, info )
call ppequ( ap, s [,scond] [,amax] [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix *A* in packed storage and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors such that *s*(*i*)*s*[*i* + 1] contains



These factors are chosen so that the scaled matrix *B* with elements b_{ij}=*s*(i)**a*$_{ij}$**s*(j) has diagonal elements equal to 1.

This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

See ?laqsp auxiliary function that uses scaling factors computed by ?ppequ.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is packed in the array *ap*:

If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix *A*.

If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix *A*.

n

INTEGER. The order of matrix *A*; *n*≥ 0.

ap

REAL for sppequ

DOUBLE PRECISION for dppequ

COMPLEX for cppequ

DOUBLE COMPLEX for zppequ.

Array, size at least max(1,*n*(*n*+1)/2). The array *ap* contains the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *packed storage* (see Matrix Storage Schemes).

Output Parameters

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*).

	If *info* = 0, the array *s* contains the scale factors for *A*.
scond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i).
amax	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix *A*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ppequ interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
s	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *scond*≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to SMLNUM or very close to BIGNUM, the matrix *A* should be scaled.

See Also

Error Analysis
Matrix Storage Schemes

?pbequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive-definite band matrix and reduce its condition number.

Syntax

```
call spbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
call dpbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
call cpbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
call zpbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
call pbequ( ab, s [,scond] [,amax] [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite band matrix *A* and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors such that `s(i)s[i + 1]` contains

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix *B* with elements b_{ij}=s(i)*a_{ij}*s(j) has diagonal elements equal to 1. This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

See ?laqsb auxiliary function that uses scaling factors computed by ?pbequ.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored in the array *ab*:

If *uplo* = 'U', the array *ab* stores the upper triangular part of the matrix *A*.

If *uplo* = 'L', the array *ab* stores the lower triangular part of the matrix *A*.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

kd

INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.

ab

REAL for spbequ

DOUBLE PRECISION for dpbequ

COMPLEX for cpbequ

DOUBLE COMPLEX for zpbequ.

Array, size *ldab* by *.

The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *band storage* (see Matrix Storage Schemes).

The second dimension of *ab* must be at least max(1, *n*).

ldab

INTEGER. The leading dimension of the array *ab*; $ldab \geq kd + 1$.

Output Parameters

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*).

If *info* = 0, the array *s* contains the scale factors for *A*.

scond — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i).

amax — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix *A*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pbequ interface are as follows:

ab — Holds the array *A* of size (*kd*+1, *n*).

s — Holds the vector of length *n*.

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to SMLNUM or very close to BIGNUM, the matrix *A* should be scaled.

See Also

Error Analysis

Matrix Storage Schemes

?syequb

Computes row and column scaling factors intended to equilibrate a symmetric indefinite matrix and reduce its condition number.

Syntax

```
call ssyequb( uplo, n, a, lda, s, scond, amax, work, info )
call dsyequb( uplo, n, a, lda, s, scond, amax, work, info )
call csyequb( uplo, n, a, lda, s, scond, amax, work, info )
call zsyequb( uplo, n, a, lda, s, scond, amax, work, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a symmetric indefinite matrix *A* and reduce its condition number (with respect to the two-norm).

The array *s* contains the scale factors, *s*(*i*) = 1/sqrt(A(i,i)). These factors are chosen so that the scaled matrix *B* with elements *b*(*i*,*j*)=*s*(i)*a(*i*,*j*)**s*(j) has ones on the diagonal.

This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*.

n

INTEGER. The order of the matrix *A*; *n*≥ 0.

a, *work*

REAL for ssyequb

DOUBLE PRECISION for dsyequb

COMPLEX for csyequb

DOUBLE COMPLEX for zsyequb.

Array *a*: *lda* by *.

Contains the *n*-by-*n* symmetric indefinite matrix *A* whose scaling factors are to be computed. Only the diagonal elements of *A* are referenced. The second dimension of *a* must be at least max(1,*n*).

work(*) is a workspace array. The dimension of *work* is at least max(1,3**n*).

lda

INTEGER. The leading dimension of *a*; *lda*≥ max(1, *m*).

Output Parameters

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*).

If *info* = 0, the array *s* contains the scale factors for *A*.

scond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i). If *scond* ≥ 0.1, and *amax* is neither too large nor too small, it is not worth scaling by *s*.

amax

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix *A*. If *amax* is very close to SMLNUM or BIGNUM, the matrix should be scaled.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

See Also

Error Analysis

Matrix Storage Schemes

?heequb

Computes row and column scaling factors intended to equilibrate a Hermitian indefinite matrix and reduce its condition number.

Syntax

```
call cheequb( uplo, n, a, lda, s, scond, amax, work, info )
call zheequb( uplo, n, a, lda, s, scond, amax, work, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a Hermitian indefinite matrix *A* and reduce its condition number (with respect to the two-norm).

The array *s* contains the scale factors, *s*(*i*) = 1/sqrt(A(i,i)). These factors are chosen so that the scaled matrix *B* with elements *b*(*i*,*j*)=*s*(i)**a*(*i*,*j*)**s*(j) has ones on the diagonal.

This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*.

n	INTEGER. The order of the matrix *A*; *n* ≥ 0.
a, *work*	COMPLEX for cheequb DOUBLE COMPLEX for zheequb. Array *a*: size *lda* by *. Contains the *n*-by-*n* symmetric indefinite matrix *A* whose scaling factors are to be computed. Only the diagonal elements of *A* are referenced. The second dimension of *a* must be at least max(1,*n*). *work*(*) is a workspace array. The dimension of *work* is at least max(1,3*n).
lda	INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *m*).

Output Parameters

s	REAL for cheequb DOUBLE PRECISION for zheequb. Array, size (*n*). If *info* = 0, the array *s* contains the scale factors for *A*.
scond	REAL for cheequb DOUBLE PRECISION for zheequb. If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i). If *scond* ≥ 0.1, and *amax* is neither too large nor too small, it is not worth scaling by *s*.
amax	REAL for cheequb DOUBLE PRECISION for zheequb. Absolute value of the largest element of the matrix *A*. If *amax* is very close to SMLNUM or BIGNUM, the matrix should be scaled.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

See Also

Error Analysis
Matrix Storage Schemes

LAPACK Linear Equation Driver Routines

Table "Driver Routines for Solving Systems of Linear Equations" lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

Driver Routines for Solving Systems of Linear Equations

Matrix type, storage scheme	Simple Driver	Expert Driver	Expert Driver using Extra-Precise Interative Refinement
general	?gesv	?gesvx	?gesvxx
general band	?gbsv	?gbsvx	?gbsvxx
general tridiagonal	?gtsv	?gtsvx	
diagonally dominant tridiagonal	?dtsvb		
symmetric/Hermitian positive-definite	?posv	?posvx	?posvxx
symmetric/Hermitian positive-definite, storage	?ppsv	?ppsvx	
symmetric/Hermitian positive-definite, band	?pbsv	?pbsvx	
symmetric/Hermitian positive-definite, tridiagonal	?ptsv	?ptsvx	
symmetric/Hermitian indefinite	?sysv/?hesv ?sysv_rook/?sysv_rk/ ?hesv_rook/?hesv_rk ?sysv_aa/?hesv_aa	?sysvx/?hesvx	?sysvxx/?hesvxx
symmetric/Hermitian indefinite, packed storage	?spsv/?hpsv	?spsvx/?hpsvx	
complex symmetric	?sysv ?sysv_rook	?sysvx	
complex symmetric, packed storage	?spsv	?spsvx	

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex). In the description of ?gesv and ?posv routines, the ? sign stands for combined character codes ds and zc for the mixed precision subroutines.

?gesv

Computes the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides.

Syntax

```
call sgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call dgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call cgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call zgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
```

```
call dsgesv( n, nrhs, a, lda, ipiv, b, ldb, x, ldx, work, swork, iter, info )
call zcgesv( n, nrhs, a, lda, ipiv, b, ldb, x, ldx, work, swork, rwork, iter, info )
call gesv( a, b [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations `A*X = B`, where *A* is an *n*-by-*n* matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The *LU* decomposition with partial pivoting and row interchanges is used to factor *A* as `A = P*L*U`, where *P* is a permutation matrix, *L* is unit lower triangular, and *U* is upper triangular. The factored form of *A* is then used to solve the system of equations `A*X = B`.

The `dsgesv` and `zcgesv` are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (`dsgesv`) or single complex precision (`zcgesv`) and use this factorization within an iterative refinement procedure to produce a solution with double precision (`dsgesv`) / double complex precision (`zcgesv`) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision performance over double precision performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to `ilaenv` in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

```
iter > itermax
```

or for all the right-hand sides:

```
rnmr < sqrt(n)*xnrm*anrm*eps*bwdmax
```

where

- `iter` is the number of the current iteration in the iterativerefinement process
- `rnmr` is the infinity-norm of the residual
- `xnrm` is the infinity-norm of the solution
- `anrm` is the infinity-operator-norm of the matrix *A*
- `eps` is the machine epsilon returned by `dlamch` ('Epsilon').

The values `itermax` and `bwdmax` are fixed to 30 and 1.0d+00 respectively.

Input Parameters

n	`INTEGER`. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.
nrhs	`INTEGER`. The number of right-hand sides, that is, the number of columns of the matrix *B*; $nrhs \geq 0$.
a	`REAL` for `sgesv` `DOUBLE PRECISION` for `dgesv` and `dsgesv`

COMPLEX for cgesv

DOUBLE COMPLEX for zgesv and zcgesv.

The array *a*(size *lda* by *) contains the *n*-by-*n* coefficient matrix *A*.

The second dimension of *a* must be at least max(1, *n*), the second dimension of *b* at least max(1,*nrhs*).

b

REAL for sgesv

DOUBLE PRECISION for dgesv and dsgesv

COMPLEX for cgesv

DOUBLE COMPLEX for zgesv and zcgesv.

The array *b*(size *ldb* by *) contains the *n*-by-*nrhs* matrix of right hand side matrix *B*.

lda

INTEGER. The leading dimension of the array *a*; *lda*≥ max(1, *n*).

ldb

INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

ldx

INTEGER. The leading dimension of the array *x*; *ldx*≥ max(1, *n*).

work

DOUBLE PRECISION for dsgesv

DOUBLE COMPLEX for zcgesv.

Workspace array, size at least max(1,*n***nrhs*). This array is used to hold the residual vectors.

swork

REAL for dsgesv

COMPLEX for zcgesv.

Workspace array, size at least max(1,*n**(*n*+*nrhs*)). This array is used to use the single precision matrix and the right-hand sides or solutions in single precision.

rwork

DOUBLE PRECISION. Workspace array, size at least max(1,*n*).

Output Parameters

a

Overwritten by the factors *L* and *U* from the factorization of *A* = *P***L***U*; the unit diagonal elements of *L* are not stored.

If iterative refinement has been successfully used (*info*= 0 and *iter*≥ 0), then *A* is unchanged.

If double precision factorization has been used (*info*= 0 and *iter* < 0), then the array *A* contains the factors *L* and *U* from the factorization *A* = *P***L***U*; the unit diagonal elements of *L* are not stored.

b

Overwritten by the solution matrix *X* for dgesv, sgesv,zgesv,zgesv. Unchanged for dsgesv and zcgesv.

ipiv

INTEGER.

Array, size at least `max(1, n)`. The pivot indices that define the permutation matrix *P*; row *i* of the matrix was interchanged with row `ipiv(i)`. Corresponds to the single precision factorization (if `info`= 0 and `iter`≥ 0) or the double precision factorization (if `info`= 0 and `iter` < 0).

x

`DOUBLE PRECISION` for `dsgesv`

`DOUBLE COMPLEX` for `zcgesv`.

Array, size *ldx* by *nrhs*. If `info` = 0, contains the *n*-by-*nrhs* solution matrix *X*.

iter

`INTEGER`.

If `iter < 0`: iterative refinement has failed, double precision factorization has been performed

- If `iter = -1`: the routine fell back to full precision for implementation- or machine-specific reason
- If `iter = -2`: narrowing the precision induced an overflow, the routine fell back to full precision
- If `iter = -3`: failure of `sgetrf` for `dsgesv`, or `cgetrf` for `zcgesv`
- If `iter = -31`: stop the iterative refinement after the 30th iteration.

If `iter > 0`: iterative refinement has been successfully used. Returns the number of iterations.

info

`INTEGER`. If `info`=0, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, $U_{i,i}$ (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gesv` interface are as follows:

a

Holds the matrix *A* of size (`n`, `n`).

b

Holds the matrix *B* of size (`n`, `nrhs`).

ipiv

Holds the vector of length *n*.

NOTE
Fortran 95 Interface is so far not available for the mixed precision subroutines `dsgesv`/`zcgesv`.

See Also

`ilaenv`

dlamch

sgetrf

Matrix Storage Schemes

?gesvx

Computes the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call sgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
```

```
call dgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
```

```
call cgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
```

```
call zgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
```

```
call gesvx( a, b, x [,af] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c] [,ferr] [,berr]
[,rcond] [,rpvgrw] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A*X = B$, where A is an n-by-n matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gesvx performs the following steps:

1. If $fact = 'E'$, real scaling factors r and c are computed to equilibrate the system:

   $trans = 'N'$: $diag(r)*A*diag(c)*\mathrm{inv}(diag(c))*X = diag(r)*B$

   $trans = 'T'$: $(diag(r)*A*diag(c))^{T}*\mathrm{inv}(diag(r))*X = diag(c)*B$

   $trans = 'C'$: $(diag(r)*A*diag(c))^{H}*\mathrm{inv}(diag(r))*X = diag(c)*B$

   Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A is overwritten by $diag(r)*A*diag(c)$ and B by $diag(r)*B$ (if $trans='N'$) or $diag(c)*B$ (if $trans = 'T'$ or $'C'$).
2. If $fact = 'N'$ or $'E'$, the *LU* decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as $A = P*L*U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, $info = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by `diag(c)` (if `trans = 'N'`) or `diag(r)` (if `trans = 'T'` or `'C'`) so that it solves the original system before equilibration.

Input Parameters

fact

`CHARACTER*1`. Must be `'F'`, `'N'`, or `'E'`.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If `fact = 'F'`: on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not `'N'`, the matrix *A* has been equilibrated with scaling factors given by *r* and *c*.
a, *af*, and *ipiv* are not modified.

If `fact = 'N'`, the matrix *A* will be copied to *af* and factored.

If `fact = 'E'`, the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

trans

`CHARACTER*1`. Must be `'N'`, `'T'`, or `'C'`.

Specifies the form of the system of equations:

If `trans = 'N'`, the system has the form `A*X = B` (No transpose).

If `trans = 'T'`, the system has the form $A^T*X = B$ (Transpose).

If `trans = 'C'`, the system has the form $A^H*X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).

n

`INTEGER`. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs

`INTEGER`. The number of right hand sides; the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

a

`REAL` for `sgesvx`

`DOUBLE PRECISION` for `dgesvx`

`COMPLEX` for `cgesvx`

`DOUBLE COMPLEX` for `zgesvx`.

The array *a*(size *lda* by *) contains the matrix *A*. If `fact = 'F'` and *equed* is not `'N'`, then *A* must have been equilibrated by the scaling factors in *r* and/or *c*. The second dimension of *a* must be at least `max(1,n)`.

af

`REAL` for `sgesvx`

`DOUBLE PRECISION` for `dgesvx`

`COMPLEX` for `cgesvx`

`DOUBLE COMPLEX` for `zgesvx`.

The array *af* `af(size ldaf by *)` is an input argument if `fact = 'F'`. It contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization `A = P*L*U` as computed by `?getrf`. If *equed* is not `'N'`, then *af* is the factored form of the equilibrated matrix *A*. The second dimension of *af* must be at least `max(1,n)`.

b

`REAL` for `sgesvx`

`DOUBLE PRECISION` for `dgesvx`

`COMPLEX` for `cgesvx`

`DOUBLE COMPLEX` for `zgesvx`.

The array *b* `b(size ldb by *)` contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least `max(1, nrhs)`.

, *work*

`REAL` for `sgesvx`

`DOUBLE PRECISION` for `dgesvx`

`COMPLEX` for `cgesvx`

`DOUBLE COMPLEX` for `zgesvx`.

work(*) is a workspace array. The dimension of *work* must be at least `max(1,4*n)` for real flavors, and at least `max(1,2*n)` for complex flavors.

lda

`INTEGER`. The leading dimension of *a*; `lda` ≥ `max(1, n)`.

ldaf

`INTEGER`. The leading dimension of *af*; `ldaf` ≥ `max(1, n)`.

ldb

`INTEGER`. The leading dimension of *b*; `ldb` ≥ `max(1, n)`.

ipiv

`INTEGER`.

Array, size at least `max(1, n)`. The array *ipiv* is an input argument if `fact = 'F'`. It contains the pivot indices from the factorization `A = P*L*U` as computed by `?getrf`; row *i* of the matrix was interchanged with row `ipiv(i)`.

equed

`CHARACTER*1`. Must be `'N'`, `'R'`, `'C'`, or `'B'`.

equed is an input argument if `fact = 'F'`. It specifies the form of equilibration that was done:

If `equed = 'N'`, no equilibration was done (always true if `fact = 'N'`).

If `equed = 'R'`, row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If `equed = 'C'`, column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

If `equed = 'B'`, both row and column equilibration was done, that is, *A* has been replaced by `diag(r)*A*diag(c)`.

r, *c*

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Arrays: *r* (size *n*), *c* (size *n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.

If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

ldx — INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork — INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, 2*n); used in complex flavors only.

Output Parameters

x — REAL for sgesvx

DOUBLE PRECISION for dgesvx

COMPLEX for cgesvx

DOUBLE COMPLEX for zgesvx.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the *equilibrated* system is:

$diag(C)^{-1}*X$, if *trans* = 'N' and *equed* = 'C' or 'B';
$diag(R)^{-1}*X$, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'. The second dimension of *x* must be at least max(1,*nrhs*).

a — Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'. If *equed* ≠ 'N', *A* is scaled on exit as follows:

equed = 'R': *A* = *diag*(*R*)*A*

equed = 'C': *A* = *A***diag*(*c*)

equed = 'B': *A* = *diag*(*R*)**A***diag*(*c*).

af

If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization *A* = *PLU* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b

Overwritten by *diag(r)*B* if *trans* = 'N' and *equed* = 'R'or 'B';

overwritten by *diag(c)*B* if *trans* = 'T' or 'C' and *equed* = 'C' or 'B';

not changed if *equed* = 'N'.

r, *c*

These arrays are output arguments if *fact*≠'F'. See the description of *r*, *c* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the estimated forward error bound for each solution vector *x(j)* (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to *x(j)*, *ferr(j)* is an estimated upper bound for the magnitude of the largest element in *(x(j) - xtrue)* divided by the magnitude of the largest element in *x(j)*. The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the component-wise relative backward error for each solution vector *x(j)*, that is, the smallest relative change in any element of *A* or *B* that makes *x(j)* an exact solution.

ipiv

If *fact* = 'N'or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A* = *P*L*U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed

If *fact*≠'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work, *rwork*

On exit, *work(1)* for real flavors, or *rwork(1)* for complex flavors (the Fortran interface) contains the reciprocal pivot growth factor norm(*A*)/norm(*U*). The "max absolute element" norm is used. If

	work(1) for real flavors, or *rwork*(1) for complex flavors is much less than 1, then the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *x*, condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with $0 < info \le n$, then *work*(1) for real flavors, or *rwork*(1) for complex flavors contains the reciprocal pivot growth factor for the leading *info* columns of *A*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, and $i \le n$, then *U*(*i*, *i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gesvx interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
af	Holds the matrix *AF* of size (*n*, *n*).
ipiv	Holds the vector of length *n*.
r	Holds the vector of length *n*. Default value for each element is *r*(*i*) = 1.0_WP.
c	Holds the vector of length *n*. Default value for each element is *c*(*i*) = 1.0_WP.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
fact	Must be 'N', 'E', or 'F'. The default value is 'N'. If *fact* = 'F', then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
equed	Must be 'N', 'B', 'C', or 'R'. The default value is 'N'.

rpvgrw	Real value that contains the reciprocal pivot growth factor norm(*A*)/ norm(*U*).

See Also

Matrix Storage Schemes

?gesvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides

Syntax

```
call sgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
iwork, info )
```

```
call dgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
iwork, info )
```

```
call cgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

```
call zgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* matrix, the columns of the matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error (`O(eps)`, where `eps` is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with `O(eps)` errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine `?gesvxx` performs the following steps:

1. If *fact* = 'E', scaling factors *r* and *c* are computed to equilibrate the system:

trans = 'N': *diag*(*r*)**A***diag*(*c*)*inv(*diag*(*c*))**X* = *diag*(*r*)**B*

trans = 'T': $(diag(r)*A*diag(c))^{T}$*inv(*diag*(*r*))**X* = *diag*(*c*)**B*

trans = 'C': $(diag(r)*A*diag(c))^{H}$*inv(*diag*(*r*))**X* = *diag*(*c*)**B*

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(r)*A*diag(c)* and *B* by *diag(r)*B* (if *trans*='N') or *diag(c)*B* (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as *A* = *P*L*U*, where *P* is a permutation matrix, *L* is a unit lower triangular matrix, and *U* is upper triangular.
3. If some $U_{i,i}$= 0, so that *U* is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A* (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for *X* and compute error bounds.
4. The system of equations is solved for *X* using the factored form of *A*.
5. By default, unless params(la_linrx_itref_i) is set to zero, the routine applies iterative refinement to improve the computed solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix *X* is premultiplied by *diag(c)* (if *trans* = 'N') or *diag(r)* (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F', on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*. Parameters *a*, *af*, and *ipiv* are not modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.

If *fact* = 'E', the matrix *A* will be equilibrated, if necessary, copied to *af* and factored.

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form *A*X* = *B* (No transpose).

If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose).

If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).

n

INTEGER. The number of linear equations; the order of the matrix *A*; *n*≥ 0.

nrhs

INTEGER. The number of right hand sides; the number of columns of the matrices *B* and *X*; *nrhs*≥ 0.

a, *af*, *b*, *work*

REAL for sgesvxx

DOUBLE PRECISION for dgesvxx

COMPLEX for cgesvxx

DOUBLE COMPLEX for zgesvxx.

Arrays: *a*(size *lda* by *), *af*(size *ldaf*by *), *b*(size *ldb* by *), *work*(*).

The array *a* contains the matrix *A*. If *fact* = 'F' and *equed* is not 'N', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*. The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F'. It contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization *A* = *P*L*U* as computed by ?getrf. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *A*. The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,4**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

lda — INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ldaf — INTEGER. The leading dimension of *af*; *ldaf* ≥ max(1, *n*).

ipiv — INTEGER.

Array, size at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization *A* = *P*L*U* as computed by ?getrf; row *i* of the matrix was interchanged with row *ipiv*(*i*).

equed — CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

If *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*r*)**A**diag*(*c*).

r, *c* — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r* (size *n*), *c* (size *n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.

If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R'or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

Each element of *r* or *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb — INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

ldx — INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).

n_err_bnds — INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

nparams — INTEGER. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.

params — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size max(1, *nparams*). Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10.0

	Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, 3*n); used in complex flavors only.

Output Parameters

x REAL for sgesvxx

DOUBLE PRECISION for dgesvxx

COMPLEX for cgesvxx

DOUBLE COMPLEX for zgesvxx.

Array, size (*ldx*,*).

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed*≠'N', and the solution to the *equilibrated* system is:

inv(*diag*(*c*))*X, if *trans* = 'N' and *equed* = 'C' or 'B'; or inv(*diag*(*r*))*X, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'. The second dimension of *x* must be at least max(1,*nrhs*).

a Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *equed*≠'N', *A* is scaled on exit as follows:

equed = 'R': *A* = *diag*(*r*)*A*

equed = 'C': *A* = *A***diag*(*c*)

equed = 'B': *A* = *diag*(*r*)**A***diag*(*c*).

af If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization *A* = *PLU* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b Overwritten by *diag*(*r*)**B* if *trans* = 'N' and *equed* = 'R' or 'B';

overwritten by *trans* = 'T' or 'C' and *equed* = 'C' or 'B';

not changed if *equed* = 'N'.

r, *c*

These arrays are output arguments if *fact*≠'F'. Each element of these arrays is a power of the radix. See the description of *r*, *c* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equlibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with 0 < *info*≤*n*, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*. In ?gesvx, this quantity is returned in *work(1)*.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the componentwise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

err_bnds_norm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left| Xtrue_{ji} - X_{ji} \right|}{\max_j \left| X_{ji} \right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt(n)*slamch(ε)* for single precision flavors and *sqrt(n)*dlamch(ε)* for double precision flavors.
err=2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt(n)*slamch(ε)* for single precision flavors and *sqrt(n)*dlamch(ε)* for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt(n)*slamch(ε)* for single precision flavors and *sqrt(n)*dlamch(ε)* for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are: $\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$ Let *z=s*a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|Xtrue_{ji} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for

each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.
err=2	"Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are: $\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$ Let *z*=*s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1.

ipiv If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A* = *P*L*U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed If *fact*≠'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

params If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified

info INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If 0 < *info*≤*n*: *U*(*info*,*info*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params(3)* = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that *err_bnds_norm*(*j*,1) = 0.0 or *err_bnds_comp*(*j*,1) = 0.0. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?gbsv

Computes the solution to the system of linear equations with a band coefficient matrix A and multiple right-hand sides.

Syntax

```
call sgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call zgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call gbsv( ab, b [,kl] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the real or complex system of linear equations *A***X* = *B*, where *A* is an *n*-by-*n* band matrix with *kl* subdiagonals and *ku* superdiagonals, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The *LU* decomposition with partial pivoting and row interchanges is used to factor *A* as *A* = *L***U*, where *L* is a product of permutation and unit lower triangular matrices with *kl* subdiagonals, and *U* is upper triangular with *kl*+*ku* superdiagonals. The factored form of *A* is then used to solve the system of equations *A***X* = *B*.

Input Parameters

n	INTEGER. The order of *A*. The number of rows in *B*; *n*≥ 0.
kl	INTEGER. The number of subdiagonals within the band of *A*; *kl*≥ 0.
ku	INTEGER. The number of superdiagonals within the band of *A*; *ku*≥ 0.

nrhs — INTEGER. The number of right-hand sides. The number of columns in *B*; *nrhs*≥ 0.

ab, *b* — REAL for sgbsv

DOUBLE PRECISION for dgbsv

COMPLEX for cgbsv

DOUBLE COMPLEX for zgbsv.

Arrays: *ab*(size *ldab* by *), *b*(size *ldb* by *).

The array *ab* contains the matrix *A* in band storage (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

ldab — INTEGER. The leading dimension of the array *ab*. (*ldab*≥ 2*kl* + *ku* +1)

ldb — INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

Output Parameters

ab — Overwritten by *L* and *U*. The diagonal and *kl* + *ku* superdiagonals of *U* are stored in the first 1 + *kl* + *ku* rows of *ab*. The multipliers used to form *L* are stored in the next *kl* rows.

b — Overwritten by the solution matrix *X*.

ipiv — INTEGER.

Array, size at least max(1, *n*). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, *U*(*i*, *i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gbsv interface are as follows:

ab — Holds the array *A* of size (2*kl*+*ku*+1,*n*).

b — Holds the matrix *B* of size (*n*,*nrhs*).

ipiv — Holds the vector of length *n*.

kl — If omitted, assumed *kl* = *ku*.

ku	Restored as *ku* = *lda*-2**kl*-1.

See Also

Matrix Storage Schemes

?gbsvx

Computes the solution to the real or complex system of linear equations with a band coefficient matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call sgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call dgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call cgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )

call zgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )

call gbsvx( ab, b, x [,kl] [,afb] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c] [,ferr]
[,berr] [,rcond] [,rpvgrw] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations *A*X = B*, $A^T*X = B$, or $A^H*X = B$, where *A* is a band matrix of order *n* with *kl* subdiagonals and *ku* superdiagonals, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gbsvx performs the following steps:

1. If *fact* = 'E', real scaling factors *r* and *c* are computed to equilibrate the system:

   trans = 'N': *diag(r)*A*diag(c) *inv(diag(c))*X = diag(r)*B*

   trans = 'T': $(diag(r)*A*diag(c))^T$ *inv(diag(r))*X = diag(c)*B*

   trans = 'C': $(diag(r)*A*diag(c))^H$ *inv(diag(r))*X = diag(c)*B*

   Whether the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(r)*A*diag(c)* and *B* by *diag(r)*B* (if *trans*='N') or *diag(c)*B* (if *trans* = 'T'or 'C').
2. If *fact* = 'N'or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as *A* = *L*U*, where *L* is a product of permutation and unit lower triangular matrices with *kl* subdiagonals, and *U* is upper triangular with *kl*+*ku* superdiagonals.
3. If some $U_{i,i}$ = 0, so that *U* is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.

4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by `diag(c)` (if `trans = 'N'`) or `diag(r)` (if `trans = 'T'` or `'C'`) so that it solves the original system before equilibration.

Input Parameters

fact

`CHARACTER*1`. Must be `'F'`, `'N'`, or `'E'`.

Specifies whether the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If `fact = 'F'`: on entry, *afb* and *ipiv* contain the factored form of *A*. If *equed* is not `'N'`, the matrix *A* is equilibrated with scaling factors given by *r* and *c*.
ab, *afb*, and *ipiv* are not modified.

If `fact = 'N'`, the matrix *A* will be copied to *afb* and factored.

If `fact = 'E'`, the matrix *A* will be equilibrated if necessary, then copied to *afb* and factored.

trans

`CHARACTER*1`. Must be `'N'`, `'T'`, or `'C'`.

Specifies the form of the system of equations:

If `trans = 'N'`, the system has the form `A*X = B` (No transpose).

If `trans = 'T'`, the system has the form $A^T*X = B$ (Transpose).

If `trans = 'C'`, the system has the form $A^H*X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).

n

`INTEGER`. The number of linear equations, the order of the matrix *A*; $n \geq 0$.

kl

`INTEGER`. The number of subdiagonals within the band of *A*; $kl \geq 0$.

ku

`INTEGER`. The number of superdiagonals within the band of *A*; $ku \geq 0$.

nrhs

`INTEGER`. The number of right hand sides, the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

ab, *afb*, *b*, *work*

`REAL` for `sgbsvx`

`DOUBLE PRECISION` for `dgbsvx`

`COMPLEX` for `cgbsvx`

`DOUBLE COMPLEX` for `zgbsvx`.

Arrays: `ab(ldab,*)`, `afb(ldafb,*)`, `b(size ldb by *)`, `work(*)`.

The array *ab* contains the matrix *A* in band storage (see Matrix Storage Schemes). The second dimension of *ab* must be at least `max(1, n)`. If `fact = 'F'` and *equed* is not `'N'`, then *A* must have been equilibrated by the scaling factors in *r* and/or *c*.

The array *afb* is an input argument if `fact = 'F'`. The second dimension of *afb* must be at least `max(1,n)`. It contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization

*A = P*L*U* as computed by ?gbtrf. *U* is stored as an upper triangular band matrix with *kl* + *ku* superdiagonals in the first 1 + *kl* + *ku* rows of *afb*. The multipliers used during the factorization are stored in the next *kl* rows. If *equed* is not 'N', then *afb* is the factored form of the equilibrated matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,3**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

ldab — INTEGER. The leading dimension of *ab*; *ldab*≥*kl*+*ku*+1.

ldafb — INTEGER. The leading dimension of *afb*; *ldafb*≥ 2**kl*+*ku*+1.

ldb — INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

ipiv — INTEGER.

Array, size at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization *A* = *P*L*U* as computed by ?gbtrf; row *i* of the matrix was interchanged with row *ipiv*(*i*).

equed — CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

if *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*r*)**A***diag*(*c*).

r, *c* — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r* (size *n*), *c* (size *n*).

The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.

If *equed* = 'R'or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C'or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N'or 'R', *c* is not accessed.

If *fact* = 'F' and *equed* = 'C'or 'B', each element of *c* must be positive.

ldx — INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).

iwork — INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, *n*); used in complex flavors only.

Output Parameters

x — REAL for sgbsvx

DOUBLE PRECISION for dgbsvx

COMPLEX for cgbsvx

DOUBLE COMPLEX for zgbsvx.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed*≠'N', and the solution to the *equilibrated* system is: inv(*diag*(*c*))***X*, if *trans* = 'N' and *equed* = 'C'or 'B'; inv(*diag*(*r*))***X*, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.

The second dimension of *x* must be at least max(1,*nrhs*).

ab — Array *ab* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *equed*≠'N', *A* is scaled on exit as follows:

equed = 'R': *A* = *diag*(*r*)***A*

equed = 'C': *A* = *A***diag*(*c*)

equed = 'B': *A* = *diag*(*r*)***A***diag*(*c*).

afb — If *fact* = 'N' or 'E', then *afb* is an output argument and on exit returns details of the *LU* factorization of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *ab* for the form of the equilibrated matrix.

b — Overwritten by *diag*(*r*)***b* if *trans* = 'N' and *equed* = 'R' or 'B';

overwritten by *diag*(*c*)***b* if *trans* = 'T' or 'C' and *equed* = 'C' or 'B';

not changed if *equed* = 'N'.

r, *c* — These arrays are output arguments if *fact*≠'F'. See the description of *r*, *c* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done).

If *rcond* is less than the machine precision (in particular, if *rcond* =0), the matrix is singular to working precision. This condition is indicated by a return code of *info*>0.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the estimated forward error bound for each solution vector *x*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to *x*(*j*), *ferr*(*j*) is an estimated upper bound for the magnitude of the largest element in (*x*(*j*) - *xtrue*) divided by the magnitude of the largest element in *x*(*j*). The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the component-wise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

ipiv

If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A* = *L***U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and *i*≤*n*, then *U*(*i*, *i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and *i* = *n*+1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

equed

If *fact*≠'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work, *rwork*

On exit, *work*(1) for real flavors, or *rwork*(1) for complex flavors, contains the reciprocal pivot growth factor norm(*A*)/norm(*U*). The "max absolute element" norm is used. If *work*(1) for real flavors, or *rwork*(1) for complex flavors is much less than 1, then the stability of

the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *x*, condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with `0 < info≤n`, then `work(1)` for real flavors, or `rwork(1)` for complex flavors contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

info — `INTEGER`. If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, and `i≤n`, then *U*(*i*, *i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If `info = i`, and `i = n+1`, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gbsvx` interface are as follows:

ab — Holds the array *A* of size (`kl+ku+1`, `n`).

b — Holds the matrix *B* of size (`n`, `nrhs`).

x — Holds the matrix *X* of size (`n`, `nrhs`).

afb — Holds the array *AF* of size (`2*kl+ku+1`, `n`).

ipiv — Holds the vector of length *n*.

r — Holds the vector of length *n*. Default value for each element is `r(i) = 1.0_WP`.

c — Holds the vector of length *n*. Default value for each element is `c(i) = 1.0_WP`.

ferr — Holds the vector of length (*nrhs*).

berr — Holds the vector of length (*nrhs*).

trans — Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.

equed — Must be `'N'`, `'B'`, `'C'`, or `'R'`. The default value is `'N'`.

fact — Must be `'N'`, `'E'`, or `'F'`. The default value is `'N'`. If `fact = 'F'`, then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

rpvgrw — Real value that contains the reciprocal pivot growth factor norm(*A*)/ norm(*U*).

kl	If omitted, assumed *kl = ku*.
ku	Restored as *ku = lda-kl-1*.

See Also

Matrix Storage Schemes

?gbsvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a banded coefficient matrix A and multiple right-hand sides

Syntax

```
call sgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams,
params, work, iwork, info )
```

```
call dgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams,
params, work, iwork, info )
```

```
call cgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams,
params, work, rwork, info )
```

```
call zgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams,
params, work, rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A*X = B$, $A^T*X = B$, or $A^H*X = B$, where *A* is an *n*-by-*n* banded matrix, the columns of the matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error (O(eps), where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with O(eps) errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?gbsvxx performs the following steps:

1. If *fact* = 'E', scaling factors *r* and *c* are computed to equilibrate the system:

trans = 'N': *diag(r)*A*diag(c)*inv(diag(c))*X = diag(r)*B*

trans = 'T': $(diag(r)*A*diag(c))^T*inv(diag(r))*X = diag(c)*B$

trans = 'C': $(diag(r)*A*diag(c))^H*inv(diag(r))*X = diag(c)*B$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(r)*A*diag(c)* and *B* by *diag(r)*B* (if *trans*='N') or *diag(c)*B* (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as *A* = *P*L*U*, where *P* is a permutation matrix, *L* is a unit lower triangular matrix, and *U* is upper triangular.
3. If some $U_{i,i}$= 0, so that *U* is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A* (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for *X* and compute error bounds.
4. The system of equations is solved for *X* using the factored form of *A*.
5. By default, unless params(1) is set to zero, the routine applies iterative refinement to improve the computed solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix *X* is premultiplied by *diag(c)* (if *trans* = 'N') or *diag(r)* (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F', on entry, *afb* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*. Parameters *ab*, *afb*, and *ipiv* are not modified.

If *fact* = 'N', the matrix *A* will be copied to *afb* and factored.

If *fact* = 'E', the matrix *A* will be equilibrated, if necessary, copied to *afb* and factored.

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form *A*X* = *B* (No transpose).

If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose).

If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).

n

INTEGER. The number of linear equations; the order of the matrix *A*; *n*≥ 0.

kl

INTEGER. The number of subdiagonals within the band of *A*; *kl*≥ 0.

ku

INTEGER. The number of superdiagonals within the band of *A*; *ku*≥ 0.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; *nrhs*≥ 0.

ab, *afb*, *b*, *work*

REAL for sgbsvxx

DOUBLE PRECISION for dgbsvxx

COMPLEX for cgbsvxx

DOUBLE COMPLEX for zgbsvxx.

Arrays: *ab*(*ldab*,*), *afb*(*ldafb*,*), *b*(size *ldb* by *), *work*(*).

The array *ab* contains the matrix *A* in band storage, in rows 1 to *kl* +*ku*+1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

ab(*ku*+1+*i*-*j*,*j*) = *A*(*i*,*j*) for max(1,*j*-*ku*) ≤*i*≤ min(*n*,*j*+*kl*).

If *fact* = 'F' and *equed* is not 'N', then *AB* must have been equilibrated by the scaling factors in *r* and/or *c*. The second dimension of *a* must be at least max(1,*n*).

The array *afb* is an input argument if *fact* = 'F'. It contains the factored form of the banded matrix *A*, that is, the factors *L* and *U* from the factorization *A* = *P*L*U* as computed by ?gbtrf. *U* is stored as an upper triangular banded matrix with *kl* + *ku* superdiagonals in rows 1 to *kl* + *ku* + 1. The multipliers used during the factorization are stored in rows *kl* + *ku* + 2 to 2**kl* + *ku* + 1. If *equed* is not 'N', then *afb* is the factored form of the equilibrated matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,4**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

ldab — INTEGER. The leading dimension of the array *ab*; *ldab*≥*kl*+*ku*+1.

ldafb — INTEGER. The leading dimension of the array *afb*; *ldafb*≥ 2**kl*+*ku* +1.

ipiv — INTEGER.

Array, size at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization *A* = *P*L*U* as computed by ?gbtrf; row *i* of the matrix was interchanged with row *ipiv*(*i*).

equed — CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

If *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*r*)**A**diag*(*c*).

r, *c*

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r* (size *n*), *c* (size *n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.

If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N'or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

Each element of *r* or *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

ldx

INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).

n_err_bnds

INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

nparams

INTEGER. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10.0
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, 2**n*); used in complex flavors only.

Output Parameters

x REAL for sgbsvxx

DOUBLE PRECISION for dgbsvxx

COMPLEX for cgbsvxx

DOUBLE COMPLEX for zgbsvxx.

Array, size *ldx* by *.

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed*≠'N', and the solution to the *equilibrated* system is:

inv(*diag*(*c*))**X*, if *trans* = 'N' and *equed* = 'C' or 'B'; or inv(*diag*(*r*))**X*, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'. The second dimension of *x* must be at least max(1,*nrhs*).

ab Array *ab* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *equed*≠'N', *A* is scaled on exit as follows:

equed = 'R': *A* = *diag*(*r*)**A*

equed = 'C': *A* = *A***diag*(*c*)

equed = 'B': *A* = *diag*(*r*)**A***diag*(*c*).

afb

If *fact* = 'N' or 'E', then *afb* is an output argument and on exit returns the factors *L* and *U* from the factorization *A* = *PLU* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

b

Overwritten by *diag*(*r*)**B* if *trans* = 'N' and *equed* = 'R' or 'B';

overwritten by *trans* = 'T' or 'C' and *equed* = 'C' or 'B';

not changed if *equed* = 'N'.

r, *c*

These arrays are output arguments if *fact*≠'F'. Each element of these arrays is a power of the radix. See the description of *r*, *c* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equlibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with 0 < *info*≤*n*, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*. In ?gbsvx, this quantity is returned in *work(1)*.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the componentwise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

err_bnds_norm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left| Xtrue_{ji} - X_{ji} \right|}{\max_j \left| X_{ji} \right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.
err=2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are: $\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$ Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|Xtrue_{ji} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1 — "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.

err=2 — "Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3 — Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let *z=s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1.

ipiv

If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A = P*L*U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed	If *fact*≠'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).
params	If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.
info	INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed. If *info* = -*i*, the *i*-th parameter had an illegal value. If 0 < *info*≤*n*: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params(3)* = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that *err_bnds_norm*(*j*,1) = 0.0 or *err_bnds_comp*(*j*,1) = 0.0. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?gtsv

Computes the solution to the system of linear equations with a tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

```
call sgtsv( n, nrhs, dl, d, du, b, ldb, info )
call dgtsv( n, nrhs, dl, d, du, b, ldb, info )
call cgtsv( n, nrhs, dl, d, du, b, ldb, info )
call zgtsv( n, nrhs, dl, d, du, b, ldb, info )
call gtsv( dl, d, du, b [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations *A*X* = *B*, where *A* is an *n*-by-*n* tridiagonal matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions. The routine uses Gaussian elimination with partial pivoting.

Note that the equation $A^T*X = B$ may be solved by interchanging the order of the arguments *du* and *dl*.

Input Parameters

n	INTEGER. The order of *A*, the number of rows in *B*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.
dl	REAL for sgtsv DOUBLE PRECISION for dgtsv COMPLEX for cgtsv DOUBLE COMPLEX for zgtsv. The array *dl* (size *n* - 1) contains the $(n - 1)$ subdiagonal elements of *A*.
d	REAL for sgtsv DOUBLE PRECISION for dgtsv COMPLEX for cgtsv DOUBLE COMPLEX for zgtsv. The array *d* (size *n*) contains the diagonal elements of *A*.
du	REAL for sgtsv DOUBLE PRECISION for dgtsv COMPLEX for cgtsv DOUBLE COMPLEX for zgtsv. The array *du* (size *n* - 1) contains the $(n - 1)$ superdiagonal elements of *A*.
b	REAL for sgtsv DOUBLE PRECISION for dgtsv COMPLEX for cgtsv DOUBLE COMPLEX for zgtsv. The array *b* (size *ldb* by *) contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldb	INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

dl	Overwritten by the $(n-2)$ elements of the second superdiagonal of the upper triangular matrix *U* from the *LU* factorization of A. These elements are stored in *dl*(1), ..., *dl*($n - 2$).
d	Overwritten by the *n* diagonal elements of *U*.
du	Overwritten by the $(n-1)$ elements of the first superdiagonal of *U*.
b	Overwritten by the solution matrix *X*.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, *U*(*i*, *i*) is exactly zero, and the solution has not been computed. The factorization has not been completed unless *i* = *n*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gtsv interface are as follows:

dl	Holds the vector of length (*n*-1).
d	Holds the vector of length *n*.
dl	Holds the vector of length (*n*-1).
b	Holds the matrix *B* of size (*n*,*nrhs*).

See Also

Matrix Storage Schemes

?gtsvx

Computes the solution to the real or complex system of linear equations with a tridiagonal coefficient matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call sgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
```

```
call dgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
```

```
call cgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
```

```
call zgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
```

```
call gtsvx( dl, d, du, b, x [,dlf] [,df] [,duf] [,du2] [,ipiv] [,fact] [,trans] [,ferr]
[,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations *A*X = B*, $A^T*X = B$, or $A^H*X = B$, where *A* is a tridiagonal matrix of order *n*, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gtsvx performs the following steps:

1. If *fact* = 'N', the *LU* decomposition is used to factor the matrix *A* as *A* = *L***U*, where *L* is a product of permutation and unit lower bidiagonal matrices and *U* is an upper triangular matrix with nonzeroes in only the main diagonal and first two superdiagonals.
2. If some $U_{i,i}$= 0, so that *U* is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
3. The system of equations is solved for *X* using the factored form of *A*.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix *A* has been supplied on entry.

If *fact* = 'F': on entry, *dlf*, *df*, *duf*, *du2*, and *ipiv* contain the factored form of *A*; arrays *dl*, *d*, *du*, *dlf*, *df*, *duf*, *du2*, and *ipiv* will not be modified.

If *fact* = 'N', the matrix *A* will be copied to *dlf*, *df*, and *duf* and factored.

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form *A***X* = *B* (No transpose).

If *trans* = 'T', the system has the form A^T**X* = *B* (Transpose).

If *trans* = 'C', the system has the form A^H**X* = *B* (Conjugate transpose).

n

INTEGER. The number of linear equations, the order of the matrix *A*; *n*≥ 0.

nrhs

INTEGER. The number of right hand sides, the number of columns of the matrices *B* and *X*; *nrhs*≥ 0.

dl,*d*,*du*,*dlf*,*df*, *duf*,*du2*,*b*,*work*

REAL for sgtsvx

DOUBLE PRECISION for dgtsvx

COMPLEX for cgtsvx

DOUBLE COMPLEX for zgtsvx.

Arrays:

dl, size (*n* -1), contains the subdiagonal elements of *A*.

d, size (*n*), contains the diagonal elements of *A*.

du, size (*n* -1), contains the superdiagonal elements of *A*.

dlf, size (*n* -1). If *fact* = 'F', then *dlf* is an input argument and on entry contains the (*n* -1) multipliers that define the matrix *L* from the *LU* factorization of *A* as computed by ?gttrf.

df, size (*n*). If *fact* = 'F', then *df* is an input argument and on entry contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

duf, size (*n* -1). If *fact* = 'F', then *duf* is an input argument and on entry contains the (*n* -1) elements of the first superdiagonal of *U*.

du2, size (*n* -2). If *fact* = 'F', then *du2* is an input argument and on entry contains the (*n*-2) elements of the second superdiagonal of *U*.

b(*ldb*, *) contains the right-hand side matrix *B*. The second dimension of *b* must be at least max(1, *nrhs*).

work(*) is a workspace array. The size of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

ldb — INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

ldx — INTEGER. The leading dimension of *x*; *ldx*≥ max(1, *n*).

ipiv — INTEGER.

Array, size at least max(1, *n*). If *fact* = 'F', then *ipiv* is an input argument and on entry contains the pivot indices, as returned by ?gttrf.

iwork — INTEGER. Workspace array, size (*n*). Used for real flavors only.

rwork — REAL for cgtsvx

DOUBLE PRECISION for zgtsvx.

Workspace array, size (*n*). Used for complex flavors only.

Output Parameters

x — REAL for sgtsvx

DOUBLE PRECISION for dgtsvx

COMPLEX for cgtsvx

DOUBLE COMPLEX for zgtsvx.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X*. The second dimension of *x* must be at least max(1, *nrhs*).

dlf — If *fact* = 'N', then *dlf* is an output argument and on exit contains the (*n*–1) multipliers that define the matrix *L* from the *LU* factorization of A.

df — If *fact* = 'N', then *df* is an output argument and on exit contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

duf — If *fact* = 'N', then *duf* is an output argument and on exit contains the (*n*–1) elements of the first superdiagonal of *U*.

du2

If *fact* = 'N', then *du2* is an output argument and on exit contains the (*n*-2) elements of the second superdiagonal of *U*.

ipiv

The array *ipiv* is an output argument if *fact* = 'N'and, on exit, contains the pivot indices from the factorization *A* = *L***U* ; row *i* of the matrix was interchanged with row *ipiv*(*i*). The value of *ipiv*(*i*) will always be *i* or *i*+1; *ipiv*(*i*)=*i* indicates a row interchange was not required.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* =0), the matrix is singular to working precision. This condition is indicated by a return code of *info*>0.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the estimated forward error bound for each solution vector *x*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to *x*(*j*), *ferr*(*j*) is an estimated upper bound for the magnitude of the largest element in (*x*(*j*) - *xtrue*) divided by the magnitude of the largest element in *x*(*j*). The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the component-wise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and *i*≤*n*, then *U*(*i*, *i*) is exactly zero. The factorization has not been completed unless *i* = *n*, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gtsvx` interface are as follows:

dl	Holds the vector of length (*n*-1).
d	Holds the vector of length *n*.
du	Holds the vector of length (*n*-1).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
dlf	Holds the vector of length (*n*-1).
df	Holds the vector of length *n*.
duf	Holds the vector of length (*n*-1).
du2	Holds the vector of length (*n*-2).
ipiv	Holds the vector of length *n*.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
fact	Must be `'N'` or `'F'`. The default value is `'N'`. If *fact* = `'F'`, then the arguments *dlf*, *df*, *duf*, *du2*, and *ipiv* must be present; otherwise, an error is returned.
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.

See Also

Matrix Storage Schemes

?dtsvb

Computes the solution to the system of linear equations with a diagonally dominant tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

```
call sdtsvb( n, nrhs, dl, d, du, b, ldb, info )
call ddtsvb( n, nrhs, dl, d, du, b, ldb, info )
call cdtsvb( n, nrhs, dl, d, du, b, ldb, info )
call zdtsvb( n, nrhs, dl, d, du, b, ldb, info )
call dtsvb( dl, d, du, b [, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The `?dtsvb` routine solves a system of linear equations *A*X = B* for *X*, where *A* is an *n*-by-*n* diagonally dominant tridiagonal matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions. The routine uses the BABE (Burning At Both Ends) algorithm.

Note that the equation $A^T*X = B$ may be solved by interchanging the order of the arguments `du` and `dl`.

Input Parameters

`n`	`INTEGER`. The order of A, the number of rows in B; $n \geq 0$.
`nrhs`	`INTEGER`. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.

`dl`, `d`, `du`, `b`	`REAL` for `sdtsvb`
	`DOUBLE PRECISION` for `ddtsvb`
	`COMPLEX` for `cdtsvb`
	`DOUBLE COMPLEX` for `zdtsvb`.
	Arrays: `dl` (size `n` - 1), `d` (size `n`), `du` (size `n` - 1), `b`(size `ldb`,*).
	The array `dl` contains the `(n - 1)` subdiagonal elements of A.
	The array `d` contains the diagonal elements of A.
	The array `du` contains the `(n - 1)` superdiagonal elements of A.
	The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least `max(1,nrhs)`.

`ldb`	`INTEGER`. The leading dimension of `b`; `ldb` $\geq$ `max(1, n)`.

Output Parameters

`dl`	Overwritten by the `(n-1)` elements of the subdiagonal of the lower triangular matrices L_1, L_2 from the factorization of A (see dttrfb).
`d`	Overwritten by the `n` diagonal element reciprocals of U.
`b`	Overwritten by the solution matrix X.
`info`	`INTEGER`. If `info = 0`, the execution is successful. If `info = -i`, the i-th parameter had an illegal value. If `info = i`, u_{ii} is exactly zero, and the solution has not been computed. The factorization has not been completed unless `i = n`.

Application Notes

A diagonally dominant tridiagonal system is defined such that $|d_i| > |dl_{i-1}| + |du_i|$ for any i:

$1 < i < n$, and $|d_1| > |du_1|$, $|d_n| > |dl_{n-1}|$

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems have no numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see `?gtsv`). The diagonally dominant systems are much faster than the canonical systems.

NOTE

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
- Applying the `?dtsvb` factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.

?posv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix A and multiple right-hand sides.

Syntax

```
call sposv( uplo, n, nrhs, a, lda, b, ldb, info )
call dposv( uplo, n, nrhs, a, lda, b, ldb, info )
call cposv( uplo, n, nrhs, a, lda, b, ldb, info )
call zposv( uplo, n, nrhs, a, lda, b, ldb, info )
call dsposv( uplo, n, nrhs, a, lda, b, ldb, x, ldx, work, swork, iter, info )
call zcposv( uplo, n, nrhs, a, lda, b, ldb, x, ldx, work, swork, rwork, iter, info )
call posv( a, b [,uplo] [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine solves for *X* the real or complex system of linear equations `A*X = B`, where *A* is an *n*-by-*n* symmetric/Hermitian positive-definite matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The Cholesky decomposition is used to factor *A* as

`A = U`^{T}`*U` (real flavors) and `A = U`^{H}`*U` (complex flavors), if `uplo = 'U'`

or `A = L*L`^{T} (real flavors) and `A = L*L`^{H} (complex flavors), if `uplo = 'L'`,

where *U* is an upper triangular matrix and *L* is a lower triangular matrix. The factored form of *A* is then used to solve the system of equations `A*X = B`.

The `dsposv` and `zcposv` are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (`dsposv`) or single complex precision (`zcposv`) and use this factorization within an iterative refinement procedure to produce a solution with double precision (`dsposv`) / double complex precision (`zcposv`) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision/complex performance over double precision/double complex performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to `ilaenv` in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if
`iter > itermax`
or for all the right-hand sides:
`rnmr < sqrt(n)*xnrm*anrm*eps*bwdmax`,
where

- `iter` is the number of the current iteration in the iterative refinement process
- `rnmr` is the infinity-norm of the residual
- `xnrm` is the infinity-norm of the solution
- `anrm` is the infinity-operator-norm of the matrix *A*
- `eps` is the machine epsilon returned by `dlamch` ('Epsilon').

The values `itermax` and `bwdmax` are fixed to 30 and 1.0d+00 respectively.

Input Parameters

uplo
`CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of *A* is stored:
If *uplo* = `'U'`, the upper triangle of *A* is stored.
If *uplo* = `'L'`, the lower triangle of *A* is stored.

n
`INTEGER`. The order of matrix *A*; *n*≥ 0.

nrhs
`INTEGER`. The number of right-hand sides, the number of columns in *B*; *nrhs*≥ 0.

a, *b*
`REAL` for `sposv`
`DOUBLE PRECISION` for `dposv` and `dsposv`.
`COMPLEX` for `cposv`
`DOUBLE COMPLEX` for `zposv` and `zcposv`.
Arrays: *a*(size *lda*,*), *b*(*ldb*, *). The array *a* contains the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least `max(1, n)`.
Note that in the case of `zcposv` the imaginary parts of the diagonal elements need not be set and are assumed to be zero.
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least `max(1,nrhs)`.

lda
`INTEGER`. The leading dimension of *a*; *lda*≥ `max(1, n)`.

ldb
`INTEGER`. The leading dimension of *b*; *ldb*≥ `max(1, n)`.

ldx
`INTEGER`. The leading dimension of the array *x*; *ldx*≥ `max(1, n)`.

work
`DOUBLE PRECISION` for `dsposv`
`DOUBLE COMPLEX` for `zcposv`.
Workspace array, size `(n*nrhs)`. This array is used to hold the residual vectors.

swork
`REAL` for `dsgesv`

COMPLEX for zcgesv.

Workspace array, size (n*(n+nrhs)). This array is used to use the single precision matrix and the right-hand sides or solutions in single precision.

rwork DOUBLE PRECISION. Workspace array, size (*n*).

Output Parameters

a If *info* = 0, the upper or lower triangular part of *a* is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.

If iterative refinement has been successfully used (*info*= 0 and *iter*≥ 0), then *A* is unchanged.

If double precision factorization has been used (*info*= 0 and *iter* < 0), then the array *A* contains the factors *L* or *U* from the Cholesky factorization.

b Overwritten by the solution matrix *X*.

x DOUBLE PRECISION for dsposv

DOUBLE COMPLEX for zcposv.

Array, size *ldx* by *nrhs*. If *info* = 0, contains the *n*-by-*nrhs* solution matrix *X*.

iter INTEGER.

If *iter* < 0: iterative refinement has failed, double precision factorization has been performed

- If *iter* = -1: the routine fell back to full precision for implementation- or machine-specific reason
- If *iter* = -2: narrowing the precision induced an overflow, the routine fell back to full precision
- If *iter* = -3: failure of spotrf for dsposv, or cpotrf for zcposv
- If *iter* = -31: stop the iterative refinement after the 30th iteration.

If *iter* > 0: iterative refinement has been successfully used. Returns the number of iterations.

info INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine posv interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?posvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix A, and provides error bounds on the solution.

Syntax

```
call sposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call dposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call cposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call zposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call posvx( a, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *Cholesky* factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where *A* is a *n*-by-*n* real symmetric/Hermitian positive definite matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?posvx performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

   ```
   diag(s)*A*diag(s)*inv(diag(s))*X = diag(s)*B.
   ```

   Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(s)*A*diag(s)* and *B* by *diag(s)*B*.
2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

   $A = U^T*U$ (real), $A = U^H*U$ (complex), if *uplo* = 'U',

   or $A = L*L^T$ (real), $A = L*L^H$ (complex), if *uplo* = 'L',

   where *U* is an upper triangular matrix and *L* is a lower triangular matrix.

3. If the leading *i*-by-*i* principal minor is not positive-definite, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by *diag*(*s*) so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F': on entry, *af* contains the factored form of *A*. If *equed* = 'Y', the matrix *A* has been equilibrated with scaling factors given by *s*.
a and *af* will not be modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.

If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

a, *af*, *b*, *work*

REAL for sposvx

DOUBLE PRECISION for dposvx

COMPLEX for cposvx

DOUBLE COMPLEX for zposvx.

Arrays: *a*(size *lda* by *), *af*(size *ldaf* by *), *b*(size *ldb* by *), *work*(*).

The array *a* contains the matrix *A* as specified by *uplo*. If *fact* = 'F' and *equed* = 'Y', then *A* must have been equilibrated by the scaling factors in *s*, and *a* must contain the equilibrated matrix *diag*(*s*)*A*diag*(*s*). The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *diag(s)*A*diag(s)*. The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,3**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

lda — INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ldaf — INTEGER. The leading dimension of *af*; *ldaf* ≥ max(1, *n*).

ldb — INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

equed — CHARACTER*1. Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

if *equed* = 'N', no equilibration was done (always true if *fact* = 'N');

if *equed* = 'Y', equilibration was done, that is, *A* has been replaced by *diag(s)*A*diag(s)*.

s — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *equed* = 'N', *s* is not accessed.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

ldx — INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork — INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork — REAL for cposvx

DOUBLE PRECISION for zposvx.

Workspace array, size at least max(1, *n*); used in complex flavors only.

Output Parameters

x — REAL for sposvx

DOUBLE PRECISION for dposvx

COMPLEX for cposvx

DOUBLE COMPLEX for zposvx.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is inv(*diag*(*s*))*X. The second dimension of *x* must be at least max(1,*nrhs*).

a

Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *fact* = 'E' and *equed* = 'Y', *A* is overwritten by *diag*(*s*)*A*diag*(*s*).

af

If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A=U^{T}*U$ or $A=L*L^{T}$ (real routines), $A=U^{H}*U$ or $A=L*L^{H}$ (complex routines) of the original matrix *A* (if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b

Overwritten by *diag*(*s*)*B, if *equed* = 'Y'; not changed if *equed* = 'N'.

s

This array is an output argument if *fact*≠'F'. See the description of *s* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* =0), the matrix is singular to working precision. This condition is indicated by a return code of *info*>0.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the estimated forward error bound for each solution vector x_j (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to x_j, *ferr*(*j*) is an estimated upper bound for the magnitude of the largest element in (x_j) - *xtrue*) divided by the magnitude of the largest element in x_j. The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the component-wise relative backward error for each solution vector x_j, that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

equed	If *fact*≠'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, and *i*≤*n*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* =0 is returned. If *info* = *i*, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine posvx interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
af	Holds the matrix *AF* of size (*n*,*n*).
s	Holds the vector of length *n*. Default value for each element is *s*(*i*) = 1.0_WP.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
fact	Must be 'N', 'E', or 'F'. The default value is 'N'. If *fact* = 'F', then *af* must be present; otherwise, an error is returned.
equed	Must be 'N' or 'Y'. The default value is 'N'.

See Also

Matrix Storage Schemes

?posvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix A applying the Cholesky factorization.

Syntax

call sposvxx(*fact*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *equed*, *s*, *b*, *ldb*, *x*, *ldx*, *rcond*, *rpvgrw*, *berr*, *n_err_bnds*, *err_bnds_norm*, *err_bnds_comp*, *nparams*, *params*, *work*, *iwork*, *info*)

call dposvxx(*fact*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *equed*, *s*, *b*, *ldb*, *x*, *ldx*, *rcond*, *rpvgrw*, *berr*, *n_err_bnds*, *err_bnds_norm*, *err_bnds_comp*, *nparams*, *params*, *work*, *iwork*, *info*)

call cposvxx(*fact*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *equed*, *s*, *b*, *ldb*, *x*, *ldx*, *rcond*, *rpvgrw*, *berr*, *n_err_bnds*, *err_bnds_norm*, *err_bnds_comp*, *nparams*, *params*, *work*, *rwork*, *info*)

call zposvxx(*fact*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *equed*, *s*, *b*, *ldb*, *x*, *ldx*, *rcond*, *rpvgrw*, *berr*, *n_err_bnds*, *err_bnds_norm*, *err_bnds_comp*, *nparams*, *params*, *work*, *rwork*, *info*)

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *Cholesky* factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is an n-by-n real symmetric/Hermitian positive definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error (O(eps), where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with O(eps) errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?posvxx performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

   diag(*s*)***A***diag*(*s*) *inv(*diag*(*s*))***X* = *diag*(*s*)***B*

   Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A is overwritten by *diag*(*s*)***A***diag*(*s*) and B by *diag*(*s*)***B*.
2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

   $A = U^T*U$ (real), $A = U^H*U$ (complex), if *uplo* = 'U',

   or $A = L*L^T$ (real), $A = L*L^H$ (complex), if *uplo* = 'L',

   where U is an upper triangular matrix and L is a lower triangular matrix.
3. If the leading i-by-i principal minor is not positive-definite, the routine returns with *info* = *i*. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A.

5. By default, unless `params(1)` is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix *X* is premultiplied by *diag(s)* so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F', on entry, *af* contains the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *s*. Parameters *a* and *af* are not modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.

If *fact* = 'E', the matrix *A* will be equilibrated, if necessary, copied to *af* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n

INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

a, *af*, *b*, *work*

REAL for sposvxx

DOUBLE PRECISION for dposvxx

COMPLEX for cposvxx

DOUBLE COMPLEX for zposvxx.

Arrays: *a*(size *lda* by *), *af*(size *ldaf*by *), *b*(size *ldb* by *), *work*(*).

The array *a* contains the matrix *A* as specified by *uplo* . If *fact* = 'F' and *equed* = 'Y', then *A* must have been equilibrated by the scaling factors in *s*, and *a* must contain the equilibrated matrix *diag(s)*A*diag(s)*. The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *diag(s)*A*diag(s)*. The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

	work(*) is a workspace array. The dimension of *work* must be at least max(1,4**n*) for real flavors, and at least max(1,2**n*) for complex flavors.
lda	INTEGER. The leading dimension of the array *a*; *lda*≥ max(1,*n*).
ldaf	INTEGER. The leading dimension of the array *af*; *ldaf*≥ max(1,*n*).
equed	CHARACTER*1. Must be 'N' or 'Y'. *equed* is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done: If *equed* = 'N', no equilibration was done (always true if *fact* = 'N'). if *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*s*)**A***diag*(*s*).
s	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument. If *equed* = 'N', *s* is not accessed. If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive. Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.
ldb	INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).
ldx	INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).
n_err_bnds	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in the *Output Arguments* section below.
nparams	INTEGER. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.
params	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are

used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10.0
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, 3*n); used in complex flavors only.

Output Parameters

x REAL for sposvxx

DOUBLE PRECISION for dposvxx

COMPLEX for cposvxx

DOUBLE COMPLEX for zposvxx.

Array, size *ldx* by *.

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed*≠'N', and the solution to the equilibrated system is:

inv(*diag*(*s*))*X.

a

Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *fact* = 'E' and *equed* = 'Y', *A* is overwritten by *diag(s)*A*diag(s)*.

af

If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A=U^{T}*U$ or $A=L*L^{T}$ (real routines), $A=U^{H}*U$ or $A=L*L^{H}$ (complex routines) of the original matrix *A* (if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b

If *equed* = 'N', *B* is not modified.

If *equed* = 'Y', *B* is overwritten by *diag(s)*B*.

s

This array is an output argument if *fact*≠'F'. Each element of this array is a power of the radix. See the description of *s* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equlibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \le n$, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the componentwise relative backward error for each solution vector *x(j)*, that is, the smallest relative change in any element of *A* or *B* that makes *x(j)* an exact solution.

err_bnds_norm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left| Xtrue_{ji} - X_{ji} \right|}{\max_j \left| X_{ji} \right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.
err=2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are: $\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$ Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1 — "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.

err=2 — "Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3 — Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\left\|Z\right\|_\infty \cdot \left\|Z^{-1}\right\|_\infty$$

Let *z*=*s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1.

equed — If *fact*≠'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

params — If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If 0 < *info*≤*n*: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params(3)* = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that *err_bnds_norm*(*j*,1) = 0.0 or *err_bnds_comp*(*j*,1) = 0.0. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?ppsv

Computes the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed coefficient matrix A and multiple right-hand sides.

Syntax

```
call sppsv( uplo, n, nrhs, ap, b, ldb, info )
call dppsv( uplo, n, nrhs, ap, b, ldb, info )
call cppsv( uplo, n, nrhs, ap, b, ldb, info )
call zppsv( uplo, n, nrhs, ap, b, ldb, info )
call ppsv( ap, b [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the real or complex system of linear equations *A***X* = *B*, where *A* is an *n*-by-*n* real symmetric/Hermitian positive-definite matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The Cholesky decomposition is used to factor *A* as

A = U^T**U* (real flavors) and *A* = U^H**U* (complex flavors), if *uplo* = 'U'

or *A* = *L**L^T (real flavors) and *A* = *L**L^H (complex flavors), if *uplo* = 'L',

where *U* is an upper triangular matrix and *L* is a lower triangular matrix. The factored form of *A* is then used to solve the system of equations *A***X* = *B*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of *A* is stored: If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.
ap, *b*	REAL for sppsv DOUBLE PRECISION for dppsv COMPLEX for cppsv DOUBLE COMPLEX for zppsv. Arrays: *ap*(size *), *b*(size *ldb*, *). The array *ap* contains the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *packed storage* (see Matrix Storage Schemes). The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldb	INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

ap	If *info* = 0, the upper or lower triangular part of *A* in packed storage is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.
b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ppsv interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*,*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?ppsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed coefficient matrix A, and provides error bounds on the solution.

Syntax

```
call sppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call dppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call cppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call zppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call ppsvx( ap, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n-by-n symmetric or Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ppsvx performs the following steps:

1. If *fact* = 'E', real scaling factors s are computed to equilibrate the system:

   ```
   diag(s)*A*diag(s)*inv(diag(s))*X = diag(s)*B.
   ```

   Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A is overwritten by *diag(s)*A*diag(s)* and *B* by *diag(s)*B*.
2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

   $A = U^T*U$ (real), $A = U^H*U$ (complex), if *uplo* = 'U',

   or $A = L*L^T$ (real), $A = L*L^H$ (complex), if *uplo* = 'L',

   where U is an upper triangular matrix and L is a lower triangular matrix.
3. If the leading i-by-i principal minor is not positive-definite, then the routine returns with *info* = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = n+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by `diag(s)` so that it solves the original system before equilibration.

Input Parameters

fact

`CHARACTER*1`. Must be `'F'`, `'N'`, or `'E'`.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If `fact = 'F'`: on entry, *afp* contains the factored form of *A*. If `equed = 'Y'`, the matrix *A* has been equilibrated with scaling factors given by *s*.
ap and *afp* will not be modified.

If `fact = 'N'`, the matrix *A* will be copied to *afp* and factored.

If `fact = 'E'`, the matrix *A* will be equilibrated if necessary, then copied to *afp* and factored.

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates whether the upper or lower triangular part of *A* is stored:

If `uplo = 'U'`, the upper triangle of *A* is stored.

If `uplo = 'L'`, the lower triangle of *A* is stored.

n

`INTEGER`. The order of matrix *A*; $n \geq 0$.

nrhs

`INTEGER`. The number of right-hand sides; the number of columns in *B*; $nrhs \geq 0$.

ap, *afp*, *b*, *work*

`REAL` for `sppsvx`

`DOUBLE PRECISION` for `dppsvx`

`COMPLEX` for `cppsvx`

`DOUBLE COMPLEX` for `zppsvx`.

Arrays: (size *), *afp*(size *), *b*(size *ldb* by *), `work(*)`.

The array *ap* contains the upper or lower triangle of the original symmetric/Hermitian matrix *A* in *packed storage* (see Matrix Storage Schemes). In case when `fact = 'F'` and `equed = 'Y'`, *ap* must contain the equilibrated matrix `diag(s)*A*diag(s)`.

The array *afp* is an input argument if `fact = 'F'` and contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not `'N'`, then *afp* is the factored form of the equilibrated matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

`work(*)` is a workspace array.

The dimension of arrays *ap* and *afp* must at least `max(1, n(n+1)/2)`; the second dimension of *b* must be at least `max(1,nrhs)`; the dimension of *work* must be at least `max(1, 3*n)` for real flavors and `max(1, 2*n)` for complex flavors.

ldb — `INTEGER`. The leading dimension of *b*; *ldb*≥ `max(1, n)`.

equed — `CHARACTER*1`. Must be `'N'` or `'Y'`.

equed is an input argument if *fact* = `'F'`. It specifies the form of equilibration that was done:

if *equed* = `'N'`, no equilibration was done (always true if *fact* = `'N'`);

if *equed* = `'Y'`, equilibration was done, that is, *A* has been replaced by *diag(s)A*diag(s)*.

s — `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = `'F'` only; otherwise it is an output argument.

If *equed* = `'N'`, *s* is not accessed.

If *fact* = `'F'` and *equed* = `'Y'`, each element of *s* must be positive.

ldx — `INTEGER`. The leading dimension of the output array *x*; *ldx*≥ `max(1, n)`.

iwork — `INTEGER`. Workspace array, size at least `max(1, n)`; used in real flavors only.

rwork — `REAL` for `cppsvx`;

`DOUBLE PRECISION` for `zppsvx`.

Workspace array, size at least `max(1, n)`; used in complex flavors only.

Output Parameters

x — `REAL` for `sppsvx`

`DOUBLE PRECISION` for `dppsvx`

`COMPLEX` for `cppsvx`

`DOUBLE COMPLEX` for `zppsvx`.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that if *equed* = `'Y'`, *A* and *B* are modified on exit, and the solution to the equilibrated system is `inv(diag(s))*X`. The second dimension of *x* must be at least `max(1,nrhs)`.

ap

Array *ap* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E'and *equed* = 'N'.

If *fact* = 'E' and *equed* = 'Y', *ap* is overwritten by *diag(s)*A*diag(s)*.

afp

If *fact* = 'N'or 'E', then *afp* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ (real routines), $A=U^H*U$ or $A=L*L^H$ (complex routines) of the original matrix *A* (if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *ap* for the form of the equilibrated matrix.

b

Overwritten by *diag(s)*B*, if *equed* = 'Y'; not changed if *equed* = 'N'.

s

This array is an output argument if *fact*≠'F'. See the description of *s* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the estimated forward error bound for each solution vector *x(j)* (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to *x(j)* , *ferr(j)* is an estimated upper bound for the magnitude of the largest element in *(x(j) - xtrue)* divided by the magnitude of the largest element in *x(j)* . The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *nrhs*). Contains the component-wise relative backward error for each solution vector *x(j)* , that is, the smallest relative change in any element of *A* or *B* that makes *x(j)* an exact solution.

equed

If *fact*≠'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

info

INTEGER. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and *i*≤*n*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `ppsvx` interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
afp	Holds the matrix *AF* of size (*n**(*n*+1)/2).
s	Holds the vector of length *n*. Default value for each element is *s*(*i*) = 1.0_WP.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
fact	Must be 'N', 'E', or 'F'. The default value is 'N'. If *fact* = 'F', then *af* must be present; otherwise, an error is returned.
equed	Must be 'N' or 'Y'. The default value is 'N'.

See Also

Matrix Storage Schemes

?pbsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite band coefficient matrix A and multiple right-hand sides.

Syntax

```
call spbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
```

```
call pbsv( ab, b [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the real or complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* symmetric/Hermitian positive definite band matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The Cholesky decomposition is used to factor *A* as

$A = U^T*U$ (real flavors) and $A = U^H*U$ (complex flavors), if *uplo* = 'U'

or $A = L*L^T$ (real flavors) and $A = L*L^H$ (complex flavors), if *uplo* = 'L',

where *U* is an upper triangular band matrix and *L* is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as *A*. The factored form of *A* is then used to solve the system of equations *A*X = B*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of *A* is stored: If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of matrix *A*; $n \geq 0$.
kd	INTEGER. The number of superdiagonals of the matrix *A* if *uplo* = 'U', or the number of subdiagonals if *uplo* = 'L'; $kd \geq 0$.
nrhs	INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.
ab, *b*	REAL for spbsv DOUBLE PRECISION for dpbsv COMPLEX for cpbsv DOUBLE COMPLEX for zpbsv. Arrays: *ab*(size *ldab* by *), *b*(size *ldb* by *). The array *ab* contains the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *band storage* (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldab	INTEGER. The leading dimension of the array *ab*; $ldab \geq kd + 1$.
ldb	INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

ab	The upper or lower triangular part of *A* (in band storage) is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*, in the same storage format as *A*.
b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pbsv interface are as follows:

ab	Holds the array *A* of size (*kd*+1, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?pbsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive-definite band coefficient matrix A, and provides error bounds on the solution.

Syntax

```
call spbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
```

```
call dpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
```

```
call cpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
```

```
call zpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
```

```
call pbsvx( ab, b, x [,uplo] [,afb] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n-by-n symmetric or Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?pbsvx` performs the following steps:

1. If `fact = 'E'`, real scaling factors s are computed to equilibrate the system:

   `diag(s)*A*diag(s)*inv(diag(s))*X = diag(s)*B`.

   Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A is overwritten by `diag(s)*A*diag(s)` and B by `diag(s)*B`.
2. If `fact = 'N'` or `'E'`, the Cholesky decomposition is used to factor the matrix A (after equilibration if `fact = 'E'`) as

   $A = U^T*U$ (real), $A = U^H*U$ (complex), if `uplo = 'U'`,

   or $A = L*L^T$ (real), $A = L*L^H$ (complex), if `uplo = 'L'`,

   where U is an upper triangular band matrix and L is a lower triangular band matrix.
3. If the leading i-by-i principal minor is not positive definite, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, `info = n+1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by `diag(s)` so that it solves the original system before equilibration.

Input Parameters

fact

`CHARACTER*1`. Must be `'F'`, `'N'`, or `'E'`.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If `fact = 'F'`: on entry, *afb* contains the factored form of A. If `equed = 'Y'`, the matrix A has been equilibrated with scaling factors given by s.
ab and *afb* will not be modified.

If `fact = 'N'`, the matrix A will be copied to *afb* and factored.

If `fact = 'E'`, the matrix A will be equilibrated if necessary, then copied to *afb* and factored.

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates whether the upper or lower triangular part of A is stored:

If `uplo = 'U'`, the upper triangle of A is stored.

If `uplo = 'L'`, the lower triangle of A is stored.

n — INTEGER. The order of matrix *A*; *n*≥ 0.

kd — INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; *kd*≥ 0.

nrhs — INTEGER. The number of right-hand sides, the number of columns in *B*; *nrhs*≥ 0.

ab, *afb*, *b*, *work* — REAL for spbsvx

DOUBLE PRECISION for dpbsvx

COMPLEX for cpbsvx

DOUBLE COMPLEX for zpbsvx.

Arrays: *ab*(size *ldab* by *), *afb*(size *ldafb* by *), *b*(size *ldb* by *), *work*(*).

The array *ab* contains the upper or lower triangle of the matrix *A* in *band storage* (see Matrix Storage Schemes).

If *fact* = 'F' and *equed* = 'Y', then *ab* must contain the equilibrated matrix *diag*(*s*)***A***diag*(*s*). The second dimension of *ab* must be at least max(1, *n*).

The array *afb* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of the band matrix *A* in the same storage format as *A*. If *equed* = 'Y', then *afb* is the factored form of the equilibrated matrix *A*. The second dimension of *afb* must be at least max(1, *n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).

work(*) is a workspace array.

The dimension of *work* must be at least max(1,3**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

ldab — INTEGER. The leading dimension of *ab*; *ldab*≥*kd*+1.

ldafb — INTEGER. The leading dimension of *afb*; *ldafb*≥*kd*+1.

ldb — INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

equed — CHARACTER*1. Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

if *equed* = 'N', no equilibration was done (always true if *fact* = 'N')

if *equed* = 'Y', equilibration was done, that is, *A* has been replaced by *diag*(*s*)***A***diag*(*s*).

s — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *equed* = 'N', *s* is not accessed.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

ldx INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).

iwork INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork REAL for cpbsvx

DOUBLE PRECISION for zpbsvx.

Workspace array, size at least max(1, *n*); used in complex flavors only.

Output Parameters

x REAL for spbsvx

DOUBLE PRECISION for dpbsvx

COMPLEX for cpbsvx

DOUBLE COMPLEX for zpbsvx.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is inv(*diag*(*s*))***X*. The second dimension of *x* must be at least max(1,*nrhs*).

ab On exit, if *fact* = 'E'and *equed* = 'Y', *A* is overwritten by *diag*(*s*)***A***diag*(*s*).

afb If *fact* = 'N'or 'E', then *afb* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A=U^{T}*U$ or $A=L*L^{T}$ (real routines), $A=U^{H}*U$ or $A=L*L^{H}$ (complex routines) of the original matrix *A* (if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *ab* for the form of the equilibrated matrix.

b Overwritten by *diag*(*s*)***B*, if *equed* = 'Y'; not changed if *equed* = 'N'.

s This array is an output argument if *fact*≠'F'. See the description of *s* in *Input Arguments* section.

rcond REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size at least `max(1, nrhs)`. Contains the estimated forward error bound for each solution vector `x(j)` (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to `x(j)`, `ferr(j)` is an estimated upper bound for the magnitude of the largest element in `(x(j) - xtrue)` divided by the magnitude of the largest element in `x(j)`. The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size at least `max(1, nrhs)`. Contains the component-wise relative backward error for each solution vector `x(j)`, that is, the smallest relative change in any element of *A* or *B* that makes `x(j)` an exact solution.

equed

If `fact`≠`'F'`, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

info

`INTEGER`. If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, and `i`≤`n`, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* =0 is returned. If `info = i`, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `pbsvx` interface are as follows:

ab

Holds the array *A* of size (`kd`+1,`n`).

b

Holds the matrix *B* of size (`n`,`nrhs`).

x

Holds the matrix *X* of size (`n`,`nrhs`).

afb

Holds the array *AF* of size (`kd`+1,`n`).

s	Holds the vector with the number of elements *n*. Default value for each element is `s(i) = 1.0_WP`.
ferr	Holds the vector with the number of elements *nrhs*.
berr	Holds the vector with the number of elements *nrhs*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
fact	Must be `'N'`, `'E'`, or `'F'`. The default value is `'N'`. If `fact = 'F'`, then *af* must be present; otherwise, an error is returned.
equed	Must be `'N'` or `'Y'`. The default value is `'N'`.

See Also

Matrix Storage Schemes

?ptsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

```
call sptsv( n, nrhs, d, e, b, ldb, info )
call dptsv( n, nrhs, d, e, b, ldb, info )
call cptsv( n, nrhs, d, e, b, ldb, info )
call zptsv( n, nrhs, d, e, b, ldb, info )
call ptsv( d, e, b [,info] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine solves for *X* the real or complex system of linear equations `A*X = B`, where *A* is an *n*-by-*n* symmetric/Hermitian positive-definite tridiagonal matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

A is factored as $A = L*D*L^T$ (real flavors) or $A = L*D*L^H$ (complex flavors), and the factored form of *A* is then used to solve the system of equations `A*X = B`.

Input Parameters

n	`INTEGER`. The order of matrix *A*; $n \geq 0$.
nrhs	`INTEGER`. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.
d	`REAL` for single precision flavors `DOUBLE PRECISION` for double precision flavors.

	Array, dimension at least `max(1, n)`. Contains the diagonal elements of the tridiagonal matrix *A*.

e, *b*	`REAL` for `sptsv`
	`DOUBLE PRECISION` for `dptsv`
	`COMPLEX` for `cptsv`
	`DOUBLE COMPLEX` for `zptsv`.
	Arrays: *e* (size *n* - 1), *b*(size *ldb* by *). The array *e* contains the `(n - 1)` subdiagonal elements of *A*.
	The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least `max(1,nrhs)`.

ldb	`INTEGER`. The leading dimension of *b*; *ldb*≥ `max(1, n)`.

Output Parameters

d	Overwritten by the *n* diagonal elements of the diagonal matrix *D* from the $L*D*L^T$ (real)/ $L*D*L^H$ (complex) factorization of *A*.
e	Overwritten by the `(n - 1)` subdiagonal elements of the unit bidiagonal factor *L* from the factorization of *A*.
b	Overwritten by the solution matrix *X*.
info	`INTEGER`. If *info* `= 0`, the execution is successful. If *info* `= -i`, the *i*-th parameter had an illegal value. If *info* `= i`, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the solution has not been computed. The factorization has not been completed unless *i* `= n`.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `ptsv` interface are as follows:

d	Holds the vector of length *n*.
e	Holds the vector of length (*n*-1).
b	Holds the matrix *B* of size (*n*,*nrhs*).

See Also

Matrix Storage Schemes

?ptsvx

Uses factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite tridiagonal coefficient matrix A, and provides error bounds on the solution.

Syntax

```
call sptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work, info )
call dptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work, info )
call cptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call zptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call ptsvx( d, e, b, x [,df] [,ef] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the Cholesky factorization $A = L*D*L^T$ (real)/$A = L*D*L^H$ (complex) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n-by-n symmetric or Hermitian positive definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ptsvx performs the following steps:

1. If *fact* = 'N', the matrix A is factored as $A = L*D*L^T$ (real flavors)/$A = L*D*L^H$ (complex flavors), where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form $A = U^T*D*U$ (real flavors)/$A = U^H*D*U$ (complex flavors).
2. If the leading i-by-i principal minor is not positive-definite, then the routine returns with *info* = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = n+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A is supplied on entry.

If *fact* = 'F': on entry, *df* and *ef* contain the factored form of A. Arrays *d*, *e*, *df*, and *ef* will not be modified.

If *fact* = 'N', the matrix A will be copied to *df* and *ef*, and factored.

n

INTEGER. The order of matrix A; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in B; *nrhs* ≥ 0.

d, *df*, *rwork*

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *d* (size *n*), *df* (size *n*), *rwork*(*n*).

The array *d* contains the *n* diagonal elements of the tridiagonal matrix *A*.

The array *df* is an input argument if *fact* = 'F' and on entry contains the *n* diagonal elements of the diagonal matrix *D* from the $L*D*L^T$ (real)/ $L*D*L^H$ (complex) factorization of *A*.

The array *rwork* is a workspace array used for complex flavors only.

e,*ef*,*b*,*work*

REAL for sptsvx

DOUBLE PRECISION for dptsvx

COMPLEX for cptsvx

DOUBLE COMPLEX for zptsvx.

Arrays: *e* (size *n* -1), *ef* (size *n* -1), *b*(size *ldb*, *), *work*(*). The array *e* contains the (*n* – 1) subdiagonal elements of the tridiagonal matrix *A*.

The array *ef* is an input argument if *fact* = 'F' and on entry contains the (*n* – 1) subdiagonal elements of the unit bidiagonal factor *L* from the $L*D*L^T$ (real)/ $L*D*L^H$ (complex) factorization of *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

The array *work* is a workspace array. The dimension of *work* must be at least 2*n for real flavors, and at least *n* for complex flavors.

ldb

INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

ldx

INTEGER. The leading dimension of *x*; *ldx*≥ max(1, *n*).

Output Parameters

x

REAL for sptsvx

DOUBLE PRECISION for dptsvx

COMPLEX for cptsvx

DOUBLE COMPLEX for zptsvx.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

df, *ef*

These arrays are output arguments if *fact* = 'N'. See the description of *df*, *ef* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size at least `max(1, nrhs)`. Contains the estimated forward error bound for each solution vector `x(j)` (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to `x(j)`, `ferr(j)` is an estimated upper bound for the magnitude of the largest element in `(x(j) - xtrue)` divided by the magnitude of the largest element in `x(j)`. The estimate is as reliable as the estimate for `rcond`, and is almost always a slight overestimate of the true error.

berr

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size at least `max(1, nrhs)`. Contains the component-wise relative backward error for each solution vector `x(j)`, that is, the smallest relative change in any element of *A* or *B* that makes `x(j)` an exact solution.

info

`INTEGER`. If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, and $i \le n$, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* =0 is returned.

If `info = i`, and $i = n + 1$, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `ptsvx` interface are as follows:

d

Holds the vector of length *n*.

e

Holds the vector of length (*n*-1).

b

Holds the matrix *B* of size (*n*,*nrhs*).

x

Holds the matrix *X* of size (*n*,*nrhs*).

df

Holds the vector of length *n*.

ef	Holds the vector of length (*n*-1).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
fact	Must be 'N' or 'F'. The default value is 'N'. If *fact* = 'F', then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

See Also

Matrix Storage Schemes

?sysv

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A and multiple right-hand sides.

Syntax

```
call ssysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call dsysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call csysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zsysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call sysv( a, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the real or complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* symmetric matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U*D*U^T$ or $A = L*D*L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of *A* is then used to solve the system of equations *A*X = B*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of *A* is stored: If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B*; $nrhs \geq 0$.

a, *b*, *work*	REAL for ssysv DOUBLE PRECISION for dsysv COMPLEX for csysv DOUBLE COMPLEX for zsysv. Arrays: *a*(size *lda* by *), *b*(size *ldb* by *), *work*(*). The array *a* contains the upper or the lower triangular part of the symmetric matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*). *work* is a workspace array, dimension at least max(1,*lwork*).
lda	INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).
lwork	INTEGER. The size of the *work* array; *lwork*≥ 1. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* below for details and for the suggested value of *lwork*.

Output Parameters

a	If *info* = 0, *a* is overwritten by the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by ?sytrf.
b	If *info* = 0, *b* is overwritten by the solution matrix *X*.
ipiv	INTEGER. Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D*, as determined by ?sytrf. If *ipiv*(*i*) = *k* >0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column. If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column. If *uplo* = 'L'and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sysv` interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
ipiv	Holds the vector of length *n*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

For better performance, try using *lwork* = *n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

See Also

Matrix Storage Schemes

?sysv_aa

*Computes the solution to a system of linear equations A * X = B for symmetric matrices.*

```
call ssysv_aa(uplo, n, nrhs, A, lda, ipiv, B, ldb, work, lwork, info)
call csysv_aa(uplo, n, nrhs, A, lda, ipiv, B, ldb, work, lwork, info)
call dsysv_aa(uplo, n, nrhs, A, lda, ipiv, B, ldb, work, lwork, info)
call zsysv_aa(uplo, n, nrhs, A, lda, ipiv, B, ldb, work, lwork, info)
```

Description

The `?sysv` routine computes the solution to a complex system of linear equations A * X = B, where A is an *n*-by-*n* symmetric matrix and X and B are *n*-by-*nrhs* matrices.

Aasen's algorithm is used to factor A as A = U * T * U^T, if *uplo* = `'U'`, or A = L * T * L^T, if *uplo* = `'L'`, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is symmetric tri-diagonal. The factored form of A is then used to solve the system of equations A * X= B.

Input Parameters

uplo

CHARACTER*1

- = 'U': The upper triangle of A is stored.
- = 'L': The lower triangle of A is stored.

n

INTEGER

The number of linear equations; that is, the order of the matrix A. *n* ≥ 0.

nrhs

INTEGER

The number of right-hand sides; that is, the number of columns of the matrix B. *nrhs* ≥ 0.

A

REAL for ssysv_aa

DOUBLE PRECISION for dsysv_aa

COMPLEX for csysv_aa

COMPLEX*16 for zsysv_aa

Array, dimension (*lda*,*n*). On entry, the symmetric matrix A. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

lda

INTEGER

The leading dimension of the array *A*.*lda* ≥ max(1, *n*).

B

REAL for ssysv_aa

DOUBLE PRECISION for dsysv_aa

COMPLEX for csysv_aa

COMPLEX*16 for zsysv_aa

Array, dimension (*ldb*,*nrhs*). On entry, the *n*-by-*nrhs* right-hand side matrix B.

ldb

INTEGER

The leading dimension of the array *B*. *ldb* ≥ max(1, *n*).

lwork

INTEGER

The length of the array *work*.

If *lwork* = *-1*, a workspace query is assumed; the routine calculates only the optimal size of the work array and returns this value as the first entry of the work array, and no error message related to *lwork* is issued by XERBLA.

Output Parameters

A

REAL for ssysv_aa

DOUBLE PRECISION for dsysv_aa

COMPLEX for csysv_aa

COMPLEX*16 for zsysv_aa

On exit, if *info* = 0, the tridiagonal matrix T and the multipliers used to obtain the factor U or L from the factorization A = U*T*U^T or A = L*T*L^T as computed by ?sytrf.

ipiv — INTEGER

Array, dimension (*n*). On exit, it contains the details of the interchanges; that is, the row and column *k* of A were interchanged with the row and column ipiv(*k*).

B — REAL for ssysv_aa

DOUBLE PRECISION for dsytrs_aa

COMPLEX for csysv_aa

COMPLEX*16 for zsysv_aa

On exit, if *info* = 0, the *n*-by-*nrhs* solution matrix X.

work — REAL for ssysv_aa

DOUBLE PRECISION for dsytrs_aa

COMPLEX for csysv_aa

COMPLEX*16 for zsysv_aa

Array, dimension (MAX(1,*lwork*)). On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info — INTEGER

- = 0: Successful exit.
- < 0: If *info* = -*i*, the i^{th} argument had an illegal value.
- > 0: If *info* = *i*, D(*i*,*i*) is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution could not be computed.

?sysv_rook

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A and multiple right-hand sides.

Syntax

```
call ssysv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call dsysv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call csysv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zsysv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call sysv_rook( a, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the real or complex system of linear equations `A*X = B`, where *A* is an *n*-by-*n* symmetric matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U*D*U^T$ or $A = L*D*L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The `?sysv_rook` routine is called to compute the factorization of a complex symmetric matrix *A* using the bounded Bunch-Kaufman ("rook") diagonal pivoting method.

The factored form of *A* is then used to solve the system of equations `A*X = B`.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates whether the upper or lower triangular part of *A* is stored:

If `uplo = 'U'`, the upper triangle of *A* is stored.

If `uplo = 'L'`, the lower triangle of *A* is stored.

n

`INTEGER`. The order of matrix *A*; $n \geq 0$.

nrhs

`INTEGER`. The number of right-hand sides; the number of columns in *B*; $nrhs \geq 0$.

a, *b*, *work*

`REAL` for `ssysv_rook`

`DOUBLE PRECISION` for `dsysv_rook`

`COMPLEX` for `csysv_rook`

`DOUBLE COMPLEX` for `zsysv_rook`.

Arrays: *a*(size *lda* by *), *b*(size *ldb* by *), `work(*)`.

The array *a* contains the upper or the lower triangular part of the symmetric matrix *A* (see *uplo*). The second dimension of *a* must be at least `max(1, n)`.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least `max(1,nrhs)`.

work is a workspace array, dimension at least `max(1,lwork)`.

lda

`INTEGER`. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldb

`INTEGER`. The leading dimension of *b*; $ldb \geq \max(1, n)$.

lwork

`INTEGER`. The size of the *work* array; $lwork \geq 1$.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`. See *Application Notes* below for details and for the suggested value of *lwork*.

Output Parameters

a	If *info* = 0, *a* is overwritten by the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by sytrf_rook.
b	If *info* = 0, *b* is overwritten by the solution matrix *X*.
ipiv	INTEGER. Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D*, as determined by sytrf_rook. If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block. If *uplo* = 'U' and *ipiv*(*k*) < 0 and *ipiv*(*k* – 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* - 1 and -*ipiv*(*k* - 1) were interchanged, and $D_{k-1:k,\ k-1:k}$ is a 2-by-2 diagonal block. If *uplo* = 'L' and *ipiv*(*k*) < 0 and *ipiv*(*k* + 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* + 1 and -*ipiv*(*k* + 1) were interchanged, and $D_{k:k+1,\ k:k+1}$ is a 2-by-2 diagonal block.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sysv_rook interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
ipiv	Holds the vector of length *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?sysv_rk

*Computes the solution to system of linear equations A * X = B for SY matrices.*

```
call ssysv_rk(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, work, lwork, info)
```

```
call dsysv_rk(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, work, lwork, info)
call csysv_rk(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, work, lwork, info)
call zsysv_rk(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, work, lwork, info)
```

Description

?sysv_rk computes the solution to a real or complex system of linear equations A * X = B, where A is an *n*-by-*n* symmetric matrix and X and B are *n*-by-*nrhs* matrices.

The bounded Bunch-Kaufman (rook) diagonal pivoting method is used to factor A as A= P*U*D*(U^T)*(P^T), if *uplo* = 'U', or A= P*L*D*(L^T)*(P^T), if *uplo* = 'L', where U (or L) is unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?sytrf_rk is called to compute the factorization of a real or complex symmetric matrix. The factored form of A is then used to solve the system of equations A * X = B by calling BLAS3 routine ?sytrs_3.

Input Parameters

uplo

CHARACTER*1

Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

- = 'U': The upper triangle of A is stored.
- = 'L': The lower triangle of A is stored.

n

INTEGER

The number of linear equations; that is, the order of the matrix A. *n* ≥ 0.

nrhs

INTEGER

The number of right-hand sides; that is, the number of columns of the matrix B. *nrhs* ≥ 0.

A

REAL for ssysv_rk

DOUBLE PRECISION for dsysv_rk

COMPLEX for csysv_rk

COMPLEX*16 for zsysv_rk

Array, dimension (*lda*,*n*). On entry, the symmetric matrix A. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

lda

INTEGER

The leading dimension of the array *A*.*lda* ≥ max(1, *n*).

B

REAL for ssysv_rk

DOUBLE PRECISION for dsysv_rk

COMPLEX for csysv_rk

COMPLEX*16 for zsysv_rk

Array, dimension (*ldb*,*nrhs*). On entry, the *n*-by-*nrhs* right-hand side matrix B.

ldb INTEGER

The leading dimension of the array *B*. *ldb* ≥ max(1, *n*).

lwork INTEGER

The length of the array *work*.

If *lwork* = *-1*, a workspace query is assumed; the routine calculates only the optimal size of the *work* array for factorization stage and returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by XERBLA.

Output Parameters

A REAL for ssysv_rk

DOUBLE PRECISION for dsysv_rk

COMPLEX for csysv_rk

COMPLEX*16 for zsysv_rk

On exit, if *info* = 0, the diagonal of the block diagonal matrix D and factors U or L as computed by ?sytrf_rk:

- *Only* diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, D(*k*,*k*) = A(*k*,*k*). Superdiagonal (or subdiagonal) elements of D are stored on exit in array *e*.
- If *uplo* = 'U', factor U in the superdiagonal part of A. If *uplo* = 'L', factor L in the subdiagonal part of A. For more information, see the description of the ?sytrf_rk routine.

e REAL for ssysv_rk

DOUBLE PRECISION for dsysv_rk

COMPLEX for csysv_rk

COMPLEX*16 for zsysv_rk

Array, dimension (*n*). On exit, contains the output computed by the factorization routine ?sytrf_rk; that is, the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', e(*i*) = D(*i*–1,*i*), *i*=1:N–1, and e(1) is set to 0. If *uplo* = 'L', e(*i*) = D(*i*+1,*i*), *i*=1:N–1, and e(*n*) is set to 0.

NOTE For 1-by-1 diagonal block D(*k*), where 1 ≤ k ≤ *n*, the element e(*k*) is set to 0 in both the *uplo* = 'U' and *uplo* = 'L' cases. For more information, see the description of the?sytrf_rk routine.

ipiv INTEGER

Array, dimension (*n*). Details of the interchanges and the block structure of D, as determined by `?sytrf_rk`. For more information, see the description of the `?sytrf_rk` routine.

B

`REAL` for `ssysv_rk`

`DOUBLE PRECISION` for `dsysv_rk`

`COMPLEX` for `csysv_rk`

`COMPLEX*16` for `zsysv_rk`

On exit, if *info* = 0, the *n*-by-*nrhs* solution matrix X.

work

`REAL` for `ssysv_rk`

`DOUBLE PRECISION` for `dsysv_rk`

`COMPLEX` for `csysv_rk`

`COMPLEX*16` for `zsysv_rk`

Array, dimension (MAX(1,*lwork*)). Work array used in the factorization stage. On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info

`INTEGER`

- = 0: successful exit.
- < 0: If *info* = *-k*, the k^{th} argument had an illegal value.
- > 0: If *info* = *k*, the matrix A is singular. If *uplo* = `'U'`, column *k* in the upper triangular part of A contains all zeros. If *uplo* = `'L'`, column *k* in the lower triangular part of A contains all zeros. Therefore D(*k*,*k*) is exactly zero, and superdiagonal elements of column *k* of U (or subdiagonal elements of column *k* of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

NOTE *info* stores only the first occurrence of a singularity; any subsequent occurrence of singularity is not stored in *info* even though the factorization always completes.

?sysvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric coefficient matrix A, and provides error bounds on the solution.

Syntax

```
call ssysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, lwork, iwork, info )
```

```
call dsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, lwork, iwork, info )
```

```
call csysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, lwork, rwork, info )
```

```
call zsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, lwork, rwork, info )
call sysvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations *A*X = B*, where *A* is a *n*-by-*n* symmetric matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?sysvx performs the following steps:

1. If *fact = 'N'*, the diagonal pivoting method is used to factor the matrix *A*. The form of the factorization is $A = U*D*U^T$ or $A = L*D*L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i}$= 0, so that *D* is exactly singular, then the routine returns with *info = i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info = n*+1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
3. The system of equations is solved for *X* using the factored form of *A*.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix *A* has been supplied on entry.

If *fact = 'F'*: on entry, *af* and *ipiv* contain the factored form of *A*. Arrays *a*, *af*, and *ipiv* will not be modified.

If *fact = 'N'*, the matrix *A* will be copied to *af* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo = 'U'*, the upper triangle of *A* is stored.

If *uplo = 'L'*, the lower triangle of *A* is stored.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

a, *af*, *b*, *work*

REAL for ssysvx

DOUBLE PRECISION for dsysvx

COMPLEX for csysvx

DOUBLE COMPLEX for zsysvx.

Arrays: *a*(size *lda* by *), *af*(size *ldaf* by *), *b*(size *ldb* by *), *work*(*).

The array *a* contains the upper or the lower triangular part of the symmetric matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ?sytrf. The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).

work(*) is a workspace array, dimension at least max(1,*lwork*).

lda — INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ldaf — INTEGER. The leading dimension of *af*; *ldaf* ≥ max(1, *n*).

ldb — INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv — INTEGER.

Array, size at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by ?sytrf.

If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*–1) = –*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = –*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx — INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

lwork — INTEGER. The size of the *work* array.

If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* below for details and for the suggested value of *lwork*.

iwork — INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork — REAL for csysvx;

DOUBLE PRECISION for zsysvx.

Workspace array, size at least `max(1, n)`; used in complex flavors only.

Output Parameters

x

`REAL` for `ssysvx`

`DOUBLE PRECISION` for `dsysvx`

`COMPLEX` for `csysvx`

`DOUBLE COMPLEX` for `zsysvx`.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least `max(1,nrhs)`.

af, *ipiv*

These arrays are output arguments if *fact* = 'N'.

See the description of *af*, *ipiv* in *Input Arguments* section.

rcond

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size at least `max(1, nrhs)`. Contains the estimated forward error bound for each solution vector *x*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to *x*(*j*), *ferr*(*j*) is an estimated upper bound for the magnitude of the largest element in (*x*(*j*) - *xtrue*) divided by the magnitude of the largest element in *x*(*j*). The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size at least `max(1, nrhs)`. Contains the component-wise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

work(1)

If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

`INTEGER`. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and *i*≤*n*, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and *i* = *n* + 1, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sysvx` interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
af	Holds the matrix *AF* of size (*n*,*n*).
ipiv	Holds the vector of length *n*.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
fact	Must be `'N'` or `'F'`. The default value is `'N'`. If *fact* = `'F'`, then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

Application Notes

The value of *lwork* must be at least `max(1,m*n)`, where for real flavors *m* = 3 and for complex flavors *m* = 2. For better performance, try using *lwork* = `max(1, m*n, n*blocksize)`, where *blocksize* is the optimal block size for `?sytrf`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*`(1)`) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

See Also

Matrix Storage Schemes

?sysvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric indefinite coefficient matrix A applying the diagonal pivoting factorization.

Syntax

```
call ssysvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
iwork, info )
```

```
call dsysvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
iwork, info )
```

```
call csysvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

```
call zsysvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *diagonal pivoting* factorization to compute the solution to a real or complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* real symmetric/Hermitian matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error (O(eps), where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with O(eps) errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?sysvxx performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

   *diag(s)*A*diag(s) *inv(diag(s))*X = diag(s)*B*

   Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(s)*A*diag(s)* and *B* by *diag(s)*B*.
2. If *fact* = 'N' or 'E', the LU decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

   *A = U*D*U*^{T}, if *uplo* = 'U',

or $A = L*D*L^T$, if *uplo* = 'L',

where *U* or *L* is a product of permutation and unit upper (lower) triangular matrices, and *D* is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

3. If some $D(i,i)=0$, so that *D* is exactly singular, the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A* (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for *X* and compute error bounds.
4. The system of equations is solved for *X* using the factored form of *A*.
5. By default, unless params(1) is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix *X* is premultiplied by *diag(r)* so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F', on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *s*. Parameters *a*, *af*, and *ipiv* are not modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.

If *fact* = 'E', the matrix *A* will be equilibrated, if necessary, copied to *af* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n

INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

a, *af*, *b*, *work*

REAL for ssysvxx

DOUBLE PRECISION for dsysvxx

COMPLEX for csysvxx

DOUBLE COMPLEX for zsysvxx.

Arrays: *a*(size *lda* by *), *af*(size *ldaf* by *), *b*(size *ldb*, *), *work*(*).

The array *a* contains the symmetric matrix *A* as specified by *uplo*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A* and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower

triangular part of *a* contains the lower triangular part of the matrix *A* and the strictly upper triangular part of *a* is not referenced. The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* and *L* from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ?sytrf. The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,4**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

lda — INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1,*n*).

ldaf — INTEGER. The leading dimension of the array *af*; *ldaf* ≥ max(1,*n*).

ipiv — INTEGER.

Array, size at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D* as determined by ?sytrf. If *ipiv(k)* > 0, rows and columns *k* and *ipiv(k)* were interchanged and *D(k,k)* is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv(k)* = *ipiv(k–1)* < 0, rows and columns *k-1* and *–ipiv(k)* were interchanged and *D*(*k*–1:*k*, *k*–1:*k*) is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv(k)* = *ipiv(k+1)* < 0, rows and columns *k +1* and *–ipiv(k)* were interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

equed — CHARACTER*1. Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

if *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*s*)**A***diag*(*s*).

s — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*). The array *s* contains the scale factors for *A*. If *equed* = 'Y', *A* is multiplied on the left and right by diag(*s*).

This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb — INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

ldx — INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).

n_err_bnds — INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in the *Output Arguments* section below.

nparams — INTEGER. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.

params — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10.0
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least max(1, *n*); used in real flavors only.

rwork REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least max(1, 3**n*); used in complex flavors only.

Output Parameters

x REAL for ssysvxx

DOUBLE PRECISION for dsysvxx

COMPLEX for csysvxx

DOUBLE COMPLEX for zsysvxx.

Array, size *ldx* by *nrhs*).

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed*≠'N', and the solution to the equilibrated system is:

inv(*diag*(*s*))**X*.

a If *fact* = 'E' and *equed* = 'Y', overwritten by *diag*(*s*)**A***diag*(*s*).

af If *fact* = 'N', *af* is an output argument and on exit returns the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*D*U^{T}$ or $A = L*D*L^{T}$.

b If *equed* = 'N', *B* is not modified.

If *equed* = 'Y', *B* is overwritten by *diag*(*s*)**B*.

s This array is an output argument if *fact*≠'F'. Each element of this array is a power of the radix. See the description of *s* in *Input Arguments* section.

rcond REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equlibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with `0 < info≤n`, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

berr

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size at least `max(1, nrhs)`. Contains the componentwise relative backward error for each solution vector `x(j)`, that is, the smallest relative change in any element of *A* or *B* that makes `x(j)` an exact solution.

err_bnds_norm

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left|Xtrue_{ji} - X_{ji}\right|}{\max_j \left|X_{ji}\right|}$$

The array is indexed by the type of error information as described below. Up to three pieces of information are returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3 Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1 "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.

err=2 "Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and

`sqrt(n)*dlamch(ε)` for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err`=3

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let `z=s*(a*diag(x))`, where *x* is the solution for the current right-hand side and `s` scales each row of `a*diag(x)` by a power of the radix so all absolute row sums of *z* are approximately 1.

ipiv

If `fact = 'N'`, *ipiv* is an output argument and on exit contains details of the interchanges and the block structure *D*, as determined by `ssytrf` for single precision flavors and `dsytrf` for double precision flavors.

equed

If `fact≠'F'`, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

params

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

info

`INTEGER`. If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, the *i*-th parameter had an illegal value.

If 0 < `info`≤*n*: *U*(`info`,`info`) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; `rcond` = 0 is returned.

If `info = n+j`: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params(3) = 0.0`, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that `err_bnds_norm(j,1) = 0.0` or `err_bnds_comp(j,1) = 0.0`. See the definition of `err_bnds_norm` and `err_bnds_comp` for *err* = 1. To get information about all of the right-hand sides, check `err_bnds_norm` or `err_bnds_comp`.

See Also

Matrix Storage Schemes

?hesv

Computes the solution to the system of linear equations with a Hermitian matrix A and multiple right-hand sides.

Syntax

```
call chesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zhesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call hesv( a, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the complex system of linear equations $A*X = B$, where A is an n-by-n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U*D*U^H$ or $A = L*D*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix A, and A is factored as $U*D*U^H$.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix A, and A is factored as $L*D*L^H$.

n

INTEGER. The order of matrix A; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.

a, *b*, *work*

COMPLEX for chesv

DOUBLE COMPLEX for zhesv.

Arrays: *a*(size *lda* by *), *b*b(size *ldb* by *), *work*(*). The array *a* contains the upper or the lower triangular part of the Hermitian matrix A (see *uplo*). The second dimension of *a* must be at least max(1, *n*).

The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work is a workspace array, dimension at least max(1,*lwork*).

lda INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ldb INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

lwork INTEGER. The size of the *work* array (*lwork* ≥ 1).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* below for details and for the suggested value of *lwork*.

Output Parameters

a If *info* = 0, *a* is overwritten by the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by ?hetrf.

b If *info* = 0, *b* is overwritten by the solution matrix *X*.

ipiv INTEGER.

Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D*, as determined by ?hetrf.

If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)–th row and column of *A* was interchanged with the *m*-th row and column.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hesv interface are as follows:

a Holds the matrix *A* of size (*n*, *n*).

b Holds the matrix *B* of size (*n*, *nrhs*).

ipiv Holds the vector of length *n*.

uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork = n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

See Also

Matrix Storage Schemes

?hesv_aa

Computes the solution to system of linear equations for HE matrices.

```
call chesv_aa(uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call zhesv_aa(uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
```

Description

?hesv_aa computes the solution to a complex system of linear equations $A * X = B$, where A is an n-by-n Hermitian matrix and X and B are n-by-$nrhs$ matrices. Aasen's algorithm is used to factor A as

$A = U * T * U^H$ if *uplo* = 'U', or

$A = L * T * L^H$ if *uplo* = 'L',

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is Hermitian and tridiagonal. The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

uplo	CHARACTER*1. If *uplo* = 'U': The upper triangle of A is stored. If *uplo* = 'L': the lower triangle of A is stored.
n	INTEGER. The number of linear equations or the order of the matrix A. $n \geq 0$.
nrhs	INTEGER. The number of right hand sides or the number of columns of the matrix B. $nrhs \geq 0$.
a	COMPLEX for chesv_aa COMPLEX*16 for zhesv_aa Array of size (*lda*, *n*). On entry, the Hermitian matrix A.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*n*).

b COMPLEX for chesv_aa

COMPLEX*16 for zhesv_aa

Array of size (*ldb*, *nrhs*). On entry, the *n*-by-*nrhs* right hand side matrix *B*.

ldb INTEGER. The leading dimension of the array *b*. *ldb*≥ max(1,*n*).

lwork INTEGER. The length of *work*. *lwork*≥ max(1, 2**n*, 3**n*-2), and for best performance *lwork*≥ max(1,*n***nb*), where *nb* is the optimal blocksize for ?hetrf.

If *lwork* < *n*, TRS is done with Level BLAS 2. If *lwork*≥*n*, TRS is done with Level BLAS 3.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

a On exit, if *info* = 0, the tridiagonal matrix *T* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*T*U^H$ or $A = L*T*L^H$ as computed by ?hetrf_aa.

ipiv INTEGER . Array of size (*n*) On exit, it contains the details of the interchanges: row and column *k* of *A* were interchanged with the row and column *ipiv*(*k*).

b On exit, if *info* = 0, the *n*-by-*nrhs* solution matrix *X*.

work COMPLEX for chesv_aa

COMPLEX*16 for zhesv_aa

Array of size (max(1, *lwork*)). On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info INTEGER.

If *info* = 0: successful exit.

If *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0: if *info* = *i*, $D_{i,i}$ is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution could not be computed.

?hesv_rk

?hesv_rk *computes the solution to a system of linear equations A * X = B for Hermitian matrices.*

```
call chesv_rk(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, work, lwork, info)
call zhesv_rk(uplo, n, nrhs, A, lda, e, ipiv, B, ldb, work, lwork, info)
```

Description

?hesv_rk computes the solution to a complex system of linear equations A * X = B, where A is an *n*-by-*n* Hermitian matrix and X and B are *n*-by-*nrhs* matrices.

The bounded Bunch-Kaufman (rook) diagonal pivoting method is used to factor A as A = P*U*D*(U^H)*(P^T), if *uplo* = 'U', or A = P*L*D*(L^H)*(P^T), if *uplo* = 'L', where U (or L) is unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?hetrf_rk is called to compute the factorization of a complex Hermitian matrix. The factored form of A is then used to solve the system of equations A * X = B by calling BLAS3 routine ?hetrs_3.

Input Parameters

uplo

CHARACTER*1

Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

- = 'U': The upper triangle of A is stored.
- = 'L': The lower triangle of A is stored.

n

INTEGER

The number of linear equations; that is, the order of the matrix A. *n* ≥ 0.

nrhs

INTEGER

The number of right-hand sides; that is, the number of columns of the matrix B. *nrhs* ≥ 0.

A

COMPLEX for chesv_rk

COMPLEX*16 for zhesv_rk

Array, dimension (*lda*,*n*). On entry, the Hermitian matrix A. If *uplo* = 'U': the leading *n*-by-*n* upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If *uplo* = 'L': the leading *n*-by-*n* lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

lda

INTEGER

The leading dimension of the array *A*.*lda* ≥ max(1, *n*).

B

COMPLEX for chesv_rk

COMPLEX*16 for zhesv_rk

On entry, the *n*-by-*nrhs* right-hand side matrix B.

The second dimension of *B* must be at least max(1, *nrhs*).

ldb INTEGER

The leading dimension of the array *B*. *ldb* ≥ max(1, *n*).

lwork INTEGER

The length of the array *work*.

If *lwork* = –1, a workspace query is assumed; the routine calculates only the optimal size of the *work* array for the factorization stage and returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by XERBLA.

Output Parameters

A COMPLEX for chesv_rk

COMPLEX*16 for zhesv_rk

On exit, if *info* = 0, diagonal of the block diagonal matrix D and factors U or L as computed by ?hetrf_rk:

- *Only* diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, D(*k*,*k*) = *A*(*k*,*k*); (superdiagonal (or subdiagonal) elements of D are stored on exit in array *e*).

  —and—
- If *uplo* = 'U', factor U in the superdiagonal part of A. If *uplo* = 'L', factor L in the subdiagonal part of A.

For more information, see the description of the ?hetrf_rk routine.

e COMPLEX for chesv_rk

COMPLEX*16 for zhesv_rk

Array, dimension (*n*). On exit, contains the output computed by the factorization routine ?hetrf_rk; that is, the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks:

- If *uplo* = 'U', e(*i*) = D(*i*-1,*i*), *i*=2:N, e(1) is set to 0.
- If *uplo* = 'L', e(*i*) = D(*i*+1,*i*), *i*=1:N-1, e(*n*) is set to 0.

> **NOTE** For a 1-by-1 diagonal block D(*k*), where 1 ≤ *k* ≤ *n*, the element e(*k*) is set to 0 in both the *uplo* = 'U' and *uplo* = 'L' cases.

For more information, see the description of the ?hetrf_rk routine.

ipiv INTEGER

Array, dimension (*n*). Details of the interchanges and the block structure of D, as determined by ?hetrf_rk.

B COMPLEX for chesv_rk

COMPLEX*16 for zhesv_rk

On exit, if *info* = 0, the *n*-by-*nrhs* solution matrix X.

work COMPLEX for chesv_rk

COMPLEX*16 for zhesv_rk

Array, dimension (MAX(1,*lwork*)). Work array used in the factorization stage. On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info INTEGER

- = 0: Successful exit.
- < 0: If *info* = -*k*, the k^{th} argument had an illegal value.
- > 0: If *info* = *k*, the matrix A is singular. If *uplo* = 'U', column *k* in the upper triangular part of A contains all zeros. If *uplo* = 'L', column *k* in the lower triangular part of A contains all zeros. Therefore D(*k*,*k*) is exactly zero, and superdiagonal elements of column *k* of U (or subdiagonal elements of column *k* of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

NOTE *info* stores only the first occurrence of a singularity; any subsequent occurrence of singularity is not stored in *info* even though the factorization always completes.

?hesv_rook

Computes the solution to the system of linear equations for Hermitian matrices using the bounded Bunch-Kaufman diagonal pivoting method.

Syntax

```
call chesv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zhesv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call hesv_rook( a, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* Hermitian matrix, and *X* and *B* are *n*-by-*nrhs* matrices.

The bounded Bunch-Kaufman ("rook") diagonal pivoting method is used to factor *A* as

$A = U*D*U^H$ if *uplo* = 'U', or

$A = L*D*L^H$ if *uplo* = 'L',

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

hetrf_rook is called to compute the factorization of a complex Hermition matrix *A* using the bounded Bunch-Kaufman ("rook") diagonal pivoting method.

The factored form of *A* is then used to solve the system of equations *A***X* = *B* by calling ?HETRS_ROOK, which uses BLAS level 2 routines.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of *A* is stored: If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*. If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*.
n	INTEGER. The number of linear equations, which is the order of matrix *A*; *n*≥ 0.
nrhs	INTEGER. The number of right-hand sides, the number of columns in *B*; *nrhs*≥ 0.
a, *b*, *work*	COMPLEX for chesv_rook COMPLEX*16 for zhesv_rook. Arrays: *a*(size *lda* by *), *b*(size *ldb* by *), *work*(*). The array *a* contains the Hermitian matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced. The second dimension of *a* must be at least max(1, *n*). The array *b* contains the *n*-by-*nrhs* right hand side matrix *B*. The second dimension of *b* must be at least max(1,*nrhs*). *work* is a workspace array, dimension at least max(1,*lwork*).
lda	INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).
lwork	INTEGER. The size of the *work* array (*lwork*≥ 1). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* below for details and for the suggested value of *lwork*.

Output Parameters

a	If *info* = 0, *a* is overwritten by the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by hetrf_rook.
b	If *info* = 0, *b* is overwritten by the *n*-by-*nrhs* solution matrix *X*.

ipiv — INTEGER.

Array, size at least `max(1, n)`. Contains details of the interchanges and the block structure of *D*, as determined by `?hetrf_rook`.

- If *uplo* = 'U':

  If *ipiv*(*k*) > 0, rows and columns *k* and *ipiv*(*k*) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

  If *ipiv*(*k*) < 0 and *ipiv*(*k* – 1) < 0, rows and columns *k* and –*ipiv*(*k*) were interchanged, rows and columns *k* - 1 and –*ipiv*(*k* – 1) were interchanged, and $D_{k-1:k,\,k-1:k}$ is a 2-by-2 diagonal block.
- If *uplo* = 'L':

  If *ipiv*(*k*) > 0, rows and columns *k* and *ipiv*(*k*) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

  If *ipiv*(*k*) < 0 and *ipiv*(*k* + 1) < 0, rows and columns *k* and –*ipiv*(*k*) were interchanged, rows and columns *k* + 1 and –*ipiv*(*k* + 1) were interchanged, and $D_{k:k+1,\,k:k+1}$ is a 2-by-2 diagonal block.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, D_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hesv_rook` interface are as follows:

a — Holds the matrix *A* of size (*n*,*n*).

b — Holds the matrix *B* of size (*n*,*nrhs*).

ipiv — Holds the vector of length *n*.

uplo — Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?hesvx

Uses the diagonal pivoting factorization to compute the solution to the complex system of linear equations with a Hermitian coefficient matrix A, and provides error bounds on the solution.

Syntax

```
call chesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, lwork, rwork, info )
```

```
call zhesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, lwork, rwork, info )
```

```
call hesvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A*X = B$, where A is an n-by-n Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?hesvx performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix A. The form of the factorization is $A = U*D*U^H$ or $A = L*D*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with *info* = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = $n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If *fact* = 'F': on entry, *af* and *ipiv* contain the factored form of A. Arrays *a*, *af*, and *ipiv* are not modified.

If *fact* = 'N', the matrix A is copied to *af* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the Hermitian matrix A, and A is factored as $U*D*U^H$.

If *uplo* = 'L', the array *a* stores the lower triangular part of the Hermitian matrix A; A is factored as $L*D*L^H$.

n

INTEGER. The order of matrix A; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides, the number of columns in *B*; *nrhs*≥ 0.

a, *af*, *b*, *work* — COMPLEX for chesvx

DOUBLE COMPLEX for zhesvx.

Arrays: *a*(size *lda* by *), *af*(size *ldaf* by *), *b*(size *ldb* by *), *work*(*).

The array *a* contains the upper or the lower triangular part of the Hermitian matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F'. It contains he block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*D*U^H$ or $A = L*D*L^H$ as computed by ?hetrf. The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).

work(*) is a workspace array of dimension at least max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; *lda*≥ max(1, *n*).

ldaf — INTEGER. The leading dimension of *af*; *ldaf*≥ max(1, *n*).

ldb — INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

ipiv — INTEGER.

Array, size at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by ?hetrf.

If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U'and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L'and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)–th row and column of *A* was interchanged with the *m*-th row and column.

ldx — INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).

lwork — INTEGER. The size of the *work* array.

If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* below for details and for the suggested value of *lwork*.

rwork — REAL for chesvx

DOUBLE PRECISION for zhesvx.

Workspace array, size at least `max(1, n)`.

Output Parameters

x

`COMPLEX` for `chesvx`

`DOUBLE COMPLEX` for `zhesvx`.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least `max(1,nrhs)`.

af, *ipiv*

These arrays are output arguments if *fact* = 'N'. See the description of *af*, *ipiv* in *Input Arguments* section.

rcond

`REAL` for `chesvx`

`DOUBLE PRECISION` for `zhesvx`.

An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

`REAL` for `chesvx`

`DOUBLE PRECISION` for `zhesvx`.

Array, size at least `max(1, nrhs)`. Contains the estimated forward error bound for each solution vector *x*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to *x*(*j*), *ferr*(*j*) is an estimated upper bound for the magnitude of the largest element in (*x*(*j*) - *xtrue*) divided by the magnitude of the largest element in *x*(*j*). The estimate is as reliable as the estimate for *rcon*, and is almost always a slight overestimate of the true error.

berr

`REAL` for `chesvx`

`DOUBLE PRECISION` for `zhesvx`.

Array, size at least `max(1, nrhs)`. Contains the component-wise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

`INTEGER`. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and *i*≤*n*, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and *i* = *n* + 1, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hesvx` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
af	Holds the matrix *AF* of size (*n*, *n*).
ipiv	Holds the vector of length *n*.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
fact	Must be `'N'` or `'F'`. The default value is `'N'`. If *fact* = `'F'`, then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

Application Notes

The value of *lwork* must be at least `2*n`. For better performance, try using *lwork* = *n*blocksize*, where *blocksize* is the optimal block size for `?hetrf`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

See Also

Matrix Storage Schemes

?hesvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a Hermitian indefinite coefficient matrix A applying the diagonal pivoting factorization.

Syntax

```
call chesvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

```
call zhesvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *diagonal pivoting* factorization to compute the solution to a complex/double complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* Hermitian matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error (O(eps), where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with O(eps) errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?hesvxx performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

   *diag(s)*A*diag(s) *inv(diag(s))*X = diag(s)*B*

   Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(s)*A*diag(s)* and *B* by *diag(s)*B*.
2. If *fact* = 'N' or 'E', the LU decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

   $A = U*D*U^T$, if *uplo* = 'U',

   or $A = L*D*L^T$, if *uplo* = 'L',

   where *U* or *L* is a product of permutation and unit upper (lower) triangular matrices, and *D* is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
3. If some *D(i,i)=0*, so that *D* is exactly singular, the routine returns with *info = i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A* (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for *X* and compute error bounds.
4. The system of equations is solved for *X* using the factored form of *A*.

5. By default, unless `params(1)` is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix *X* is premultiplied by *diag(r)* so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F', on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *s*. Parameters *a*, *af*, and *ipiv* are not modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.

If *fact* = 'E', the matrix *A* will be equilibrated, if necessary, copied to *af* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n

INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

a, *af*, *b*, *work*

COMPLEX for chesvxx

DOUBLE COMPLEX for zhesvxx.

Arrays: *a*(size *lda* by *), *af*(size *ldaf* by *), *b*(*ldb*, *), *work(*)*.

The array *a* contains the Hermitian matrix *A* as specified by *uplo*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A* and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A* and the strictly upper triangular part of *a* is not referenced. The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* and *L* from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ?hetrf. The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array. The dimension of *work* must be at least max(1,5*n).

lda INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1,*n*).

ldaf INTEGER. The leading dimension of the array *af*; *ldaf* ≥ max(1,*n*).

ipiv INTEGER.

Array, size at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D* as determined by ?sytrf.

If *ipiv(k)* > 0, rows and columns *k* and *ipiv(k)* were interchanged and *D(k,k)* is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv(k)* = *ipiv(k-1)* < 0, rows and columns *k-1* and *-ipiv(k)* were interchanged and *D*(*k*-1:*k*, *k*-1:*k*) is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv(k)* = *ipiv(k+1)* < 0, rows and columns *k+1* and *-ipiv(k)* were interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

equed CHARACTER*1. Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

if *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*s*)**A***diag*(*s*).

s REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Array, size (*n*). The array *s* contains the scale factors for *A*. If *equed* = 'Y', *A* is multiplied on the left and right by diag(*s*).

This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array *b*; *ldb* ≥ max(1, *n*).

ldx INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

n_err_bnds INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in the *Output Arguments* section below.

nparams

INTEGER. Specifies the number of parameters set in *params*. If ≤ 0, the *params* array is never referenced and default values are used.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10
Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

rwork

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Workspace array, size at least max(1, 3*n); used in complex flavors only.

Output Parameters

x

COMPLEX for chesvxx

DOUBLE COMPLEX for zhesvxx.

Array, size *ldx* by *nrhs*.

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed*≠'N', and the solution to the equilibrated system is:

inv(*diag*(*s*))***X*.

a

If *fact* = 'E' and *equed* = 'Y', overwritten by *diag*(*s*)***A***diag*(*s*).

af

If *fact* = 'N', *af* is an output argument and on exit returns the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization *A* = *U***D***U*^{T} or *A* = *L***D***L*^{T}.

b

If *equed* = 'N', *B* is not modified.

If *equed* = 'Y', *B* is overwritten by *diag*(*s*)***B*.

s

This array is an output argument if *fact*≠'F'. Each element of this array is a power of the radix. See the description of *s* in *Input Arguments* section.

rcond

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equlibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with 0 < *info*≤*n*, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

berr

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Array, size at least max(1, *nrhs*). Contains the component-wise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

err_bnds_norm

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left| Xtrue_{ji} - X_{ji} \right|}{\max_j \left| X_{ji} \right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1 — "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for `chesvxx` and *sqrt*(*n*)**dlamch*(ε) for `zhesvxx`.

err=2 — "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for `chesvxx` and *sqrt*(*n*)**dlamch*(ε) for `zhesvxx`. This error bound should only be trusted if the previous boolean is true.

err=3 — Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for `chesvxx` and *sqrt*(*n*)**dlamch*(ε) for `zhesvxx` to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp

`REAL` for `chesvxx`

`DOUBLE PRECISION` for `zhesvxx`.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left| Xtrue_{ji} - X_{ji} \right|}{\left| X_{ji} \right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for chesvxx and *sqrt*(*n*)**dlamch*(ε) for zhesvxx.
err=2	"Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for chesvxx and *sqrt*(*n*)**dlamch*(ε) for zhesvxx. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for chesvxx and *sqrt*(*n*)**dlamch*(ε) for zhesvxx to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are: $\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$ Let *z=s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1.

ipiv If *fact* = 'N', *ipiv* is an output argument and on exit contains details of the interchanges and the block structure *D*, as determined by ssytrf for single precision flavors and dsytrf for double precision flavors.

equed If *fact*≠'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

params If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

info INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = *-i*, the *i*-th parameter had an illegal value.

If 0 < *info*≤*n*:*U*(*info*,*info*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params(3)* = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that *err_bnds_norm*(*j*,1) = 0.0 or *err_bnds_comp*(*j*,1) = 0.0. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?spsv

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A stored in packed format, and multiple right-hand sides.

Syntax

```
call sspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call cspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call spsv( ap, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the real or complex system of linear equations *A***X* = *B*, where *A* is an *n*-by-*n* symmetric matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U*D*U^T$ or $A = L*D*L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of *A* is then used to solve the system of equations *A***X* = *B*.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

ap, *b*

REAL for sspsv

DOUBLE PRECISION for dspsv

COMPLEX for cspsv

DOUBLE COMPLEX for zspsv.

Arrays: *ap*(size *), *b*(size *ldb* by *).

The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

ldb

INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

ap

The block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by ?sptrf, stored as a packed triangular matrix in the same storage format as *A*.

b

If *info* = 0, *b* is overwritten by the solution matrix *X*.

ipiv

INTEGER.

Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D*, as determined by ?sptrf.

If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U'and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L'and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `spsv` interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*,*nrhs*).
ipiv	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?spsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric coefficient matrix A stored in packed format, and provides error bounds on the solution.

Syntax

```
call sspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call dspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call cspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call zspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call spsvx( ap, b, x [,uplo] [,afp] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations *A*X = B*, where *A* is a *n*-by-*n* symmetric matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?spsvx` performs the following steps:

1. If *fact = 'N'*, the diagonal pivoting method is used to factor the matrix A. The form of the factorization is $A = U*D*U^T$ or $A = L*D*L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i}$= 0, so that *D* is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n*+1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
3. The system of equations is solved for *X* using the factored form of *A*.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix *A* has been supplied on entry.

If *fact* = 'F': on entry, *afp* and *ipiv* contain the factored form of *A*. Arrays *ap*, *afp*, and *ipiv* are not modified.

If *fact* = 'N', the matrix *A* is copied to *afp* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:

If *uplo* = 'U', the array *ap* stores the upper triangular part of the symmetric matrix *A*, and *A* is factored as $U*D*U^T$.

If *uplo* = 'L', the array *ap* stores the lower triangular part of the symmetric matrix *A*; *A* is factored as $L*D*L^T$.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

ap, *afp*, *b*, *work*

REAL for sspsvx

DOUBLE PRECISION for dspsvx

COMPLEX for cspsvx

DOUBLE COMPLEX for zspsvx.

Arrays: *ap*(size *), *afp*(size *), *b*(size *ldb* by *), *work*(*).

The array *ap* contains the upper or lower triangle of the symmetric matrix *A* in *packed storage* (see Matrix Storage Schemes).

The array *afp* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ?sptrf, in the same storage format as *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

work(*) is a workspace array.

The dimension of arrays *ap* and *afp* must at least `max(1, n(n+1)/2)`; the second dimension of *b* must be at least `max(1,nrhs)`; the dimension of *work* must be at least `max(1,3*n)` for real flavors and `max(1,2*n)` for complex flavors.

ldb — `INTEGER`. The leading dimension of *b*; *ldb*≥ `max(1, n)`.

ipiv — `INTEGER`.

Array, size at least `max(1, n)`. The array *ipiv* is an input argument if *fact* = `'F'`. It contains details of the interchanges and the block structure of *D*, as determined by `?sptrf`.

If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = `'U'` and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = `'L'` and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx — `INTEGER`. The leading dimension of the output array *x*; *ldx*≥ `max(1, n)`.

iwork — `INTEGER`. Workspace array, size at least `max(1, n)`; used in real flavors only.

rwork — `REAL` for `cspsvx`

`DOUBLE PRECISION` for `zspsvx`.

Workspace array, size at least `max(1, n)`; used in complex flavors only.

Output Parameters

x — `REAL` for `sspsvx`

`DOUBLE PRECISION` for `dspsvx`

`COMPLEX` for `cspsvx`

`DOUBLE COMPLEX` for `zspsvx`.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least `max(1,nrhs)`.

afp, *ipiv* — These arrays are output arguments if *fact* = `'N'`. See the description of *afp*, *ipiv* in *Input Arguments* section.

rcond — `REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

	An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if `rcond = 0`), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.
ferr, *berr*	`REAL` for single precision flavors `DOUBLE PRECISION` for double precision flavors. Arrays, size at least `max(1, nrhs)`. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
info	`INTEGER`. If `info = 0`, the execution is successful. If `info = -i`, the *i*-th parameter had an illegal value. If `info = i`, and $i \le n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If `info = i`, and *i* = *n* + 1, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `spsvx` interface are as follows:

ap	Holds the array *A* of size (`n*(n+1)/2`).
b	Holds the matrix *B* of size (`n,nrhs`).
x	Holds the matrix *X* of size (`n,nrhs`).
afp	Holds the array *AF* of size (`n*(n+1)/2`).
ipiv	Holds the vector with the number of elements *n*.
ferr	Holds the vector with the number of elements *nrhs*.
berr	Holds the vector with the number of elements *nrhs*.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
fact	Must be `'N'` or `'F'`. The default value is `'N'`. If `fact = 'F'`, then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

See Also

Matrix Storage Schemes

?hpsv

Computes the solution to the system of linear equations with a Hermitian coefficient matrix A stored in packed format, and multiple right-hand sides.

Syntax

```
call chpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call hpsv( ap, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations *A*X = B*, where *A* is an *n*-by-*n* Hermitian matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U*D*U^H$ or $A = L*D*L^H$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of *A* is then used to solve the system of equations *A*X = B*.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns in *B*; $nrhs \geq 0$.

ap, *b*

COMPLEX for chpsv

DOUBLE COMPLEX for zhpsv.

Arrays: *ap*(size *), *b*(size *ldb* by *).

The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

ldb

INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

ap	The block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by ?hptrf, stored as a packed triangular matrix in the same storage format as *A*.
b	If *info* = 0, *b* is overwritten by the solution matrix *X*.
ipiv	INTEGER. Array, size at least max(1, *n*). Contains details of the interchanges and the block structure of *D*, as determined by ?hptrf. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column. If *uplo* = 'U'and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column. If *uplo* = 'L'and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hpsv interface are as follows:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*,*nrhs*).
ipiv	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?hpsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a Hermitian coefficient matrix A stored in packed format, and provides error bounds on the solution.

Syntax

```
call chpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

```
call zhpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call hpsvx( ap, b, x [,uplo] [,afp] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A*X = B$, where A is a n-by-n Hermitian matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?hpsvx performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix A. The form of the factorization is $A = U*D*U^H$ or $A = L*D*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = *n*+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If *fact* = 'F': on entry, *afp* and *ipiv* contain the factored form of A. Arrays *ap*, *afp*, and *ipiv* are not modified.

If *fact* = 'N', the matrix A is copied to *afp* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If *uplo* = 'U', the array *ap* stores the upper triangular part of the Hermitian matrix A, and A is factored as $U*D*U^H$.

If *uplo* = 'L', the array *ap* stores the lower triangular part of the Hermitian matrix A, and A is factored as $L*D*L^H$.

n

INTEGER. The order of matrix A; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.

ap, *afp*, *b*, *work*

COMPLEX for chpsvx

DOUBLE COMPLEX for zhpsvx.

Arrays: *ap*(size *), *afp*(size *), *b*(size *ldb* by *), *work*(*).

The array *ap* contains the upper or lower triangle of the Hermitian matrix *A* in *packed storage* (see Matrix Storage Schemes).

The array *afp* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*D*U^H$ or $A = L*D*L^H$ as computed by ?hptrf, in the same storage format as *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

work(*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1,2**n*).

ldb

INTEGER. The leading dimension of *b*; *ldb*≥ max(1, *n*).

ipiv

INTEGER.

Array, size at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by ?hptrf.

If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U'and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L'and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx

INTEGER. The leading dimension of the output array *x*; *ldx*≥ max(1, *n*).

rwork

REAL for chpsvx

DOUBLE PRECISION for zhpsvx.

Workspace array, size at least max(1, *n*).

Output Parameters

x

COMPLEX for chpsvx

DOUBLE COMPLEX for zhpsvx.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

afp, *ipiv* — These arrays are output arguments if *fact* = 'N'. See the description of *afp*, *ipiv* in *Input Arguments* section.

rcond — REAL for chpsvx

DOUBLE PRECISION for zhpsvx.

An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr — REAL for chpsvx

DOUBLE PRECISION for zhpsvx.

Array, size at least max(1, *nrhs*). Contains the estimated forward error bound for each solution vector *x*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to *x*(*j*), *ferr*(*j*) is an estimated upper bound for the magnitude of the largest element in (*x*(*j*) - *xtrue*) divided by the magnitude of the largest element in *x*(*j*). The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr — REAL for chpsvx

DOUBLE PRECISION for zhpsvx.

Array, size at least max(1, *nrhs*). Contains the component-wise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and $i \le n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and *i* = *n* + 1, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hpsvx interface are as follows:

ap — Holds the array *A* of size (*n**(*n*+1)/2).

b — Holds the matrix *B* of size (*n*, *nrhs*).

x	Holds the matrix *X* of size (*n*, *nrhs*).
afp	Holds the array *AF* of size (*n**(*n*+1)/2).
ipiv	Holds the vector with the number of elements *n*.
ferr	Holds the vector with the number of elements *nrhs*.
berr	Holds the vector with the number of elements *nrhs*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
fact	Must be 'N' or 'F'. The default value is 'N'. If *fact* = 'F', then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

See Also

Matrix Storage Schemes

LAPACK Least Squares and Eigenvalue Problem Routines

This section includes descriptions of LAPACK computational routines and driver routines for solving linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. For a full reference on LAPACK routines and related information see [LUG].

Least Squares Problems. A typical *least squares problem* is as follows: given a matrix *A* and a vector *b*, find the vector *x* that minimizes the sum of squares $\Sigma_i((Ax)_i - b_i)^2$ or, equivalently, find the vector *x* that minimizes the 2-norm $||Ax - b||_2$.

In the most usual case, *A* is an *m*-by-*n* matrix with $m \geq n$ and rank(*A*) = *n*. This problem is also referred to as finding the *least squares solution* to an *overdetermined* system of linear equations (here we have more equations than unknowns). To solve this problem, you can use the *QR* factorization of the matrix A (see QR Factorization).

If $m < n$ and rank(*A*) = *m*, there exist an infinite number of solutions *x* which exactly satisfy $Ax = b$, and thus minimize the norm $||Ax - b||_2$. In this case it is often useful to find the unique solution that minimizes $||x||_2$. This problem is referred to as finding the *minimum-norm solution* to an *underdetermined* system of linear equations (here we have more unknowns than equations). To solve this problem, you can use the *LQ* factorization of the matrix *A* (see LQ Factorization).

In the general case you may have a *rank-deficient least squares problem*, with rank(*A*) < min(*m*, *n*): find the *minimum-norm least squares solution* that minimizes both $||x||_2$ and $||Ax - b||^2$. In this case (or when the rank of A is in doubt) you can use the *QR* factorization with pivoting or *singular value decomposition* (see Singular Value Decomposition).

Eigenvalue Problems. The eigenvalue problems (from German *eigen* "own") are stated as follows: given a matrix *A*, find the *eigenvalues*λ and the corresponding *eigenvectorsz* that satisfy the equation

$Az = \lambda z$ (right eigenvectors *z*)

or the equation

$z^H A = \lambda z^H$ (left eigenvectors *z*).

If *A* is a real symmetric or complex Hermitian matrix, the above two equations are equivalent, and the problem is called a *symmetric* eigenvalue problem. Routines for solving this type of problems are described in the topic Symmetric Eigenvalue Problems.

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the topic Nonsymmetric Eigenvalue Problems.

The library also includes routines that handle *generalized symmetric-definite eigenvalue problems*: find the eigenvalues λ and the corresponding eigenvectors *x* that satisfy one of the following equations:

$Az = \lambda Bz$, $ABz = \lambda z$, or $BAz = \lambda z$,

where *A* is symmetric or Hermitian, and *B* is symmetric positive-definite or Hermitian positive-definite. Routines for reducing these problems to standard symmetric eigenvalue problems are described in the topic Generalized Symmetric-Definite Eigenvalue Problems.

To solve a particular problem, you usually call several computational routines. Sometimes you need to combine the routines of this chapter with other LAPACK routines described in "LAPACK Routines: Linear Equations" as well as with BLAS routines described in "BLAS and Sparse BLAS Routines".

For example, to solve a set of least squares problems minimizing $||Ax - b||^2$ for all columns *b* of a given matrix *B* (where *A* and *B* are real matrices), you can call `?geqrf` to form the factorization $A = QR$, then call `?ormqr` to compute $C = Q^H B$ and finally call the BLAS routine `?trsm` to solve for *X* the system of equations $RX = C$.

Another way is to call an appropriate driver routine that performs several tasks in one call. For example, to solve the least squares problem the driver routine `?gels` can be used.

LAPACK Least Squares and Eigenvalue Problem Computational Routines

In the topics that follow, the descriptions of LAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

Orthogonal Factorizations

Singular Value Decomposition

Symmetric Eigenvalue Problems

Generalized Symmetric-Definite Eigenvalue Problems

Nonsymmetric Eigenvalue Problems

Generalized Nonsymmetric Eigenvalue Problems

Generalized Singular Value Decomposition

See also the respective driver routines.

Orthogonal Factorizations: LAPACK Computational Routines

This topic describes the LAPACK routines for the *QR (RQ)* and *LQ (QL)* factorization of matrices. Routines for the *RZ* factorization as well as for generalized *QR* and *RQ* factorizations are also included.

QR Factorization. Assume that *A* is an *m*-by-*n* matrix to be factored.

If $m \geq n$, the *QR* factorization is given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = (Q_1, Q_2) \begin{pmatrix} R \\ 0 \end{pmatrix}$$

where *R* is an *n*-by-*n* upper triangular matrix with real diagonal elements, and *Q* is an *m*-by-*m* orthogonal (or unitary) matrix.

You can use the *QR* factorization for solving the following least squares problem: minimize $||Ax - b||^2$ where *A* is a full-rank *m*-by-*n* matrix ($m \geq n$). After factoring the matrix, compute the solution *x* by solving $Rx = (Q_1)^T b$.

If $m < n$, the *QR* factorization is given by

$$A = QR = Q(R_1 R_2)$$

where *R* is trapezoidal, R_1 is upper triangular and R_2 is rectangular.

Q is represented as a product of min(m, n) *elementary reflectors*. Routines are provided to work with Q in this representation.

LQ Factorization LQ factorization of an m-by-n matrix A is as follows. If $m \le n$,

$$A = (L, 0)\,Q = (L, 0)\begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = (LQ_1)$$

where L is an m-by-m lower triangular matrix with real diagonal elements, and Q is an n-by-n orthogonal (or unitary) matrix.

If $m > n$, the LQ factorization is

$$A = \begin{pmatrix} L_1 \\ L_2 \end{pmatrix} Q$$

where L_1 is an n-by-n lower triangular matrix, L_2 is rectangular, and Q is an n-by-n orthogonal (or unitary) matrix.

You can use the LQ factorization to find the minimum-norm solution of an underdetermined system of linear equations $Ax = b$ where A is an m-by-n matrix of rank m $(m < n)$. After factoring the matrix, compute the solution vector x as follows: solve $Ly = b$ for y, and then compute $x = (Q_1)^H y$.

Table "Computational Routines for Orthogonal Factorization" lists LAPACK routines that perform orthogonal factorization of matrices.

Computational Routines for Orthogonal Factorization

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	geqrf	geqpf	orgqr	ormqr
	geqr	geqp3	ungqr	unmqr
	geqrfp			gemqr
general matrices, blocked QR factorization	geqrt			gemqrt
general matrices, RQ factorization	gerqf		orgrq	ormrq
			ungrq	unmrq

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, LQ factorization	gelqf		orglq	ormlq
	gelq		unglq	unmlq
				gemlq
general matrices, blocked LQ factorization	gelqt			gemlqt
general matrices, QL factorization	geqlf		orgql	ormql
			ungql	unmql
trapezoidal matrices, RZ factorization	tzrzf			ormrz
				unmrz
pair of matrices, generalized QR factorization	ggqrf			
pair of matrices, generalized RQ factorization	ggrqf			
triangular-pentagonal matrices, blocked QR factorization	tpqrt			tpmqrt
triangular-pentagonal matrices, blocked LQ factorization	tplqt			tpmlqt

?geqrf
Computes the QR factorization of a general m-by-n matrix.

Syntax

```
call sgeqrf(m, n, a, lda, tau, work, lwork, info)
call dgeqrf(m, n, a, lda, tau, work, lwork, info)
call cgeqrf(m, n, a, lda, tau, work, lwork, info)
call zgeqrf(m, n, a, lda, tau, work, lwork, info)
call geqrf(a [, tau] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the *QR* factorization of a general *m*-by-*n* matrix *A* (see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in *A* (*n*≥ 0).
a, *work*	REAL for sgeqrf DOUBLE PRECISION for dgeqrf COMPLEX for cgeqrf DOUBLE COMPLEX for zgeqrf. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array (*lwork*≥*n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the min(*m*,*n*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m*≥*n*); the elements below the diagonal, with the array *tau*, present the orthogonal matrix *Q* as a product of min(*m*,*n*) elementary reflectors (see Orthogonal Factorizations).
tau	REAL for sgeqrf DOUBLE PRECISION for dgeqrf COMPLEX for cgeqrf DOUBLE COMPLEX for zgeqrf. Array, size at least max (1, min(*m*, *n*)). Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors (see Orthogonal Factorizations).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine *geqrf* interface are the following:

`a`	Holds the matrix *A* of size (*m*,*n*).
`tau`	Holds the vector of length min(*m*,*n*)

Application Notes

For better performance, try using `lwork = n*blocksize`, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$||E||_2 = O(\varepsilon)||A||_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$	if $m = n$,
$(2/3)n^2(3m-n)$	if $m > n$,
$(2/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $||A*x - b||_2$ for all columns *b* of a given matrix *B*, you can call the following:

`?geqrf` (this routine)	to factorize $A = QR$;
ormqr	to compute $C = Q^T*B$ (for real matrices);
unmqr	to compute $C = Q^H*B$ (for complex matrices);
trsm (a BLAS routine)	to solve $R*X = C$.

(The columns of the computed *X* are the least squares solution vectors *x*.)

To compute the elements of *Q* explicitly, call

orgqr	(for real matrices)
ungqr	(for complex matrices).

See Also

mkl_progress

Matrix Storage Schemes

?geqr
Computes a QR factorization of a general matrix, with best performance for tall and skinny matrices.

```
call sgeqr(m, n, a, lda, t, tsize, work, lwork, info)
call dgeqr(m, n, a, lda, t, tsize, work, lwork, info)
call cgeqr(m, n, a, lda, t, tsize, work, lwork, info)
call zgeqr(m, n, a, lda, t, tsize, work, lwork, info)
```

Description

The ?geqr routine computes a QR factorization of an *m*-by-*n* matrix *A*. If the matrix is tall and skinny (*m* is substantially larger than *m*), a highly scalable algorithm is used to avoid communication overhead.

NOTE
The internal format of the elementary reflectors generated by ?geqr is only compatible with the ?gemqr routine and not any other QR routines.

Input Parameters

m
INTEGER. The number of rows of the matrix *A*. *m*≥ 0.

n
REAL for sgeqr
INTEGER. The number of columns of the matrix *A*. *n*≥ 0.

a
DOUBLE PRECISION for dgeqr
COMPLEX for cgeqr
COMPLEX*16 for zgeqr
Array of size (*lda*,*n*). On entry, the *m*-by-*n* matrix *A*.

lda
INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*m*).

tsize
INTEGER. If *tsize*≥ 5, the size of the array *t*. If *tsize* = -1 or *tsize* = -2, then the routine performs a workspace query. The routine calculates the sizes required for the *t* and *work* arrays and returns these values as the first entries of the *t* and *work* arrays, without issuing any error message related to *t* or *work* by xerbla.

If *tsize* = -1, the routine calculates the optimal size of *t* for optimum performance and returns this value in *t*(1).

If *tsize* = -2, the routine calculates then minimum size required for *t* and returns this value in *t*(1).

lwork

INTEGER. The size of the array *work*. If *lwork* = -1 or *lwork* = -2, then the routine performs a workspace query. The routine only calculates the sizes of the *t* and *work* arrays and returns these values as the first entries of the *t* and *work* arrays, without issuing any error message related to *t* or *work* by xerbla.

If *lwork* = -1, the routine calculates the optimal size of *work* for optimum performance and returns this value in *work*(1).

If *lwork* = -2, the routine calculates the minimum size required for *work* and returns this value in *work*(1).

Output Parameters

a

REAL for sgeqr

DOUBLE PRECISION for dgeqr

COMPLEX for cgeqr

COMPLEX*16 for zgeqr

On exit, the elements on and above the diagonal of the array contain the (min(*m*,*n*))-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m*≥*n*); the elements below the diagonal represent *Q*.

t

REAL for sgelq

DOUBLE PRECISION for dgelq

COMPLEX for cgelq

COMPLEX*16 for zgelq

Array, size (max(5,*tsize*)).

If *info* = 0, *t*(1) returns the optimal value for *tsize*. You can specify that it return the minimum required value for *tsize* instead - see the *tsize* description for details. The remaining entries of *t* contains part of the data structure used to represent *Q*. To apply or construct Q, you need to retain *a* and *t* and pass them to other routines.

work

REAL for sgelq

DOUBLE PRECISION for dgelq

COMPLEX for cgelq

COMPLEX*16 for zgelq

Array, size (max(1,*lwork*)).

If *info* = 0, *work*(1) contains the optimal value for *lwork*. You can specify that it return the minimum required value for *lwork* instead - see the *lwork* description for details.

info

INTEGER.

info = 0 indicates a successful exit.

info < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

See Also

?gemqr Multiples a matrix *C* by a real orthogonal or complex unitary matrix *Q*, as computed by ?geqr, with best performance for tall and skinny matrices.

?geqrfp
Computes the QR factorization of a general m-by-n matrix with non-negative diagonal elements.

Syntax

```
call sgeqrfp(m, n, a, lda, tau, work, lwork, info)
call dgeqrfp(m, n, a, lda, tau, work, lwork, info)
call cgeqrfp(m, n, a, lda, tau, work, lwork, info)
call zgeqrfp(m, n, a, lda, tau, work, lwork, info)
```

Include Files

- mkl.fi

Description

The routine forms the *QR* factorization of a general *m*-by-*n* matrix *A* (see Orthogonal Factorizations). No pivoting is performed. The diagonal entries of *R* are real and nonnegative.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in *A* (*n*≥ 0).
a, *work*	REAL for sgeqrfp DOUBLE PRECISION for dgeqrfp COMPLEX for cgeqrfp DOUBLE COMPLEX for zgeqrfp. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array (*lwork*≥*n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See Application Notes for the suggested value of *lwork*.

Output Parameters

a

Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the min(*m*,*n*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m*≥*n*); the elements below the diagonal, with the array *tau*, present the orthogonal matrix *Q* as a product of min(*m*,*n*) elementary reflectors (see Orthogonal Factorizations).

The diagonal elements of the matrix *R* are real and non-negative.

tau

REAL for sgeqrfp

DOUBLE PRECISION for dgeqrfp

COMPLEX for cgeqrfp

DOUBLE COMPLEX for zgeqrfp.

Array, size at least max (1, min(*m*, *n*)). Contains scalars that define elementary reflectors for the matrix *Q*in its decomposition in a product of elementary reflectors (see Orthogonal Factorizations).

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine *geqrfp* interface are the following:

a

Holds the matrix *A* of size (*m*,*n*).

tau

Holds the vector of length min(*m*,*n*)

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$||E||_2 = O(\varepsilon)||A||_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$	if $m = n$,
$(2/3)n^2(3m-n)$	if $m > n$,
$(2/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $||A*x - b||_2$ for all columns b of a given matrix B, you can call the following:

`?geqrfp` (this routine)	to factorize $A = QR$;
ormqr	to compute $C = Q^T*B$ (for real matrices);
unmqr	to compute $C = Q^H*B$ (for complex matrices);
trsm (a BLAS routine)	to solve $R*X = C$.

(The columns of the computed X are the least squares solution vectors x.)

To compute the elements of Q explicitly, call

orgqr	(for real matrices)
ungqr	(for complex matrices).

See Also

`mkl_progress`

Matrix Storage Schemes

?geqrt
Computes a blocked QR factorization of a general real or complex matrix using the compact WY representation of Q.

Syntax

```
call sgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call dgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call cgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call zgeqrt(m, n, nb, a, lda, t, ldt, work, info)
```

```
call geqrt(a, t, nb[, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The strictly lower triangular matrix *V* contains the elementary reflectors *H*(*i*) in the *i*th column below the diagonal. For example, if m=5 and n=3, the matrix *V* is

$$V = \begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

where v_i represents one of the vectors that define *H*(*i*). The vectors are returned in the lower triangular part of array a.

NOTE
The 1s along the diagonal of *V* are not stored in `a`.

Let `k = min(m,n)`. The number of blocks is `b = ceiling(k/nb)`, where each block is of order `nb` except for the last block, which is of order `ib = k - (b-1)*nb`. For each of the *b* blocks, a upper triangular block reflector factor is computed: `t1, t2, ..., tb`. The `nb`-by-`nb` (and *ib*-by-*ib* for the last block) *t*s are stored in the `nb`-by-`n` array `t` as

`t = (t1t2 ... tb)`.

Input Parameters

`m`	`INTEGER`. The number of rows in the matrix *A* (`m` ≥ 0).
`n`	`INTEGER`. The number of columns in *A* (`n` ≥ 0).
`nb`	`INTEGER`. The block size to be used in the blocked QR (min(`m`, `n`) ≥ `nb` ≥ 1).

`a`, `work`	`REAL` for `sgeqrt`
	`DOUBLE PRECISION` for `dgeqrt`
	`COMPLEX` for `cgeqrt`
	`COMPLEX*16` for `zgeqrt`.
	Arrays: `a`DIMENSION (`lda`, `n`) contains the `m`-by-`n` matrix *A*.
	`work`DIMENSION (`nb`, `n`) is a workspace array.

`lda`	`INTEGER`. The leading dimension of `a`; at least max(1, `m`).
`ldt`	`INTEGER`. The leading dimension of `t`; at least `nb`.

Output Parameters

`a`	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the min(`m`,`n`)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if `m`≥`n`); the elements below the diagonal, with the array `t`, present the orthogonal matrix *Q* as a product of min(`m`,`n`) elementary reflectors (see Orthogonal Factorizations).

`t`	`REAL` for `sgeqrt`
	`DOUBLE PRECISION` for `dgeqrt`
	`COMPLEX` for `cgeqrt`
	`COMPLEX*16` for `zgeqrt`.
	Array, `DIMENSION` (`ldt`, min(`m`, `n`)).
	The upper triangular block reflector's factors stored as a sequence of upper triangular blocks.

`info`	`INTEGER`. If `info` = 0, the execution is successful. If `info` < 0 and `info` = -*i*, the *i*th argument had an illegal value.

?gemqrt

Multiplies a general matrix by the orthogonal/unitary matrix Q of the QR factorization formed by `?geqrt`.

Syntax

```
call sgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call dgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call cgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call zgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call gemqrt( v, t, c, k, nb[, trans][, side][, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The `?gemqrt` routine overwrites the general real or complex *m*-by-*n* matrix*C* with

	side ='L'	*side* ='R'
trans = 'N':	*Q***C*	*C***Q*
trans = 'T':	Q^T**C*	*C**Q^T
trans = 'C':	Q^H**C*	*C**Q^H

where *Q* is a real orthogonal (complex unitary) matrix defined as the product of *k* elementary reflectors

Q = *H*(1) *H*(2)... *H*(*k*) = *I* - *V***T**V^T for real flavors, and

Q = *H*(1) *H*(2)... *H*(*k*) = *I* - *V***T**V^H for complex flavors,

generated using the compact WY representation as returned by geqrt. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — CHARACTER

='L': apply *Q*, Q^T, or Q^H from the left.

='R': apply *Q*, Q^T, or Q^H from the right.

trans — CHARACTER

='N', no transpose, apply *Q*.

='T', transpose, apply Q^T.

='C', transpose, apply Q^H.

m — INTEGER. The number of rows in the matrix *C*, (*m* ≥ 0).

n — INTEGER. The number of columns in the matrix *C*, (*n* ≥ 0).

k — INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m* ≥ *k*≥0

If *side* = 'R', *n* ≥ *k*≥0.

nb INTEGER.

The block size used for the storage of *t*, *k* ≥ *nb* ≥ 1. This must be the same value of *nb* used to generate *t* in geqrt.

v REAL for sgemqrt

DOUBLE PRECISION for dgemqrt

COMPLEX for cgemqrt

COMPLEX*16 for zgemqrt.

Array, DIMENSION (*ldv*, *k*).

The *i*th column must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,...,*k*, as returned by geqrt in the first *k* columns of its array argument *a*.

ldv INTEGER. The leading dimension of the array *v*.

if *side* = 'L', *ldv* must be at least max(1,*m*);

if *side* = 'R', *ldv* must be at least max(1,*n*).

t REAL for sgemqrt

DOUBLE PRECISION for dgemqrt

COMPLEX for cgemqrt

COMPLEX*16 for zgemqrt.

Array, DIMENSION (*ldt*, *k*).

The upper triangular factors of the block reflectors as returned by geqrt.

ldt INTEGER. The leading dimension of the array *t*. *ldt* must be at least *nb*.

c REAL for sgemqrt

DOUBLE PRECISION for dgemqrt

COMPLEX for cgemqrt

COMPLEX*16 for zgemqrt.

The *m*-by-*n* matrix *C*.

ldc INTEGER. The leadi*n*ng dimension of the array *c*. *ldc* must be at least max(1, *m*).

work REAL for sgemqrt

DOUBLE PRECISION for dgemqrt

COMPLEX for cgemqrt

COMPLEX*16 for zgemqrt.

Workspace array.

If *side* = 'L' DIMENSION *n***nb*.

If *side* = 'R' DIMENSION *m***nb*.

Output Parameters

c	Overwritten by the product $Q*C$, $C*Q$, Q^T*C, $C*Q^T$, Q^H*C, or $C*Q^H$ as specified by *side* and *trans*.
info	INTEGER. = 0: the execution is successful. < 0: if *info* = -*i*, the *i*th argument had an illegal value.

?geqpf
Computes the QR factorization of a general m-by-n matrix with pivoting.

Syntax

```
call sgeqpf(m, n, a, lda, jpvt, tau, work, info)
call dgeqpf(m, n, a, lda, jpvt, tau, work, info)
call cgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
call zgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
call geqpf(a, jpvt [,tau] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine is deprecated and has been replaced by routine geqp3.

The routine ?geqpf forms the *QR* factorization of a general *m*-by-*n* matrix *A* with column pivoting: *A*P = Q*R* (see Orthogonal Factorizations). Here *P* denotes an *n*-by-*n* permutation matrix.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in *A* (*n*≥ 0).
a, *work*	REAL for sgeqpf DOUBLE PRECISION for dgeqpf COMPLEX for cgeqpf DOUBLE COMPLEX for zgeqpf. Arrays: *a* (*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* (*lwork*) is a workspace array. The size of the *work* array must be at least max(1, 3*n*) for real flavors and at least max(1, *n*) for complex flavors.
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).

jpvt INTEGER. Array, size at least max(1, *n*).

On entry, if *jpvt*(*i*) > 0, the *i*-th column of *A* is moved to the beginning of *A***P* before the computation, and fixed in place during the computation.

If *jpvt*(*i*) = 0, the *i*th column of *A* is a free column (that is, it may be interchanged during the computation with any other free column).

rwork REAL for cgeqpf

DOUBLE PRECISION for zgeqpf.

A workspace array, DIMENSION at least max(1, 2**n*).

Output Parameters

a Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the min(*m*,*n*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m*≥*n*); the elements below the diagonal, with the array *tau*, present the orthogonal matrix *Q* as a product of min(*m*,*n*) elementary reflectors (see Orthogonal Factorizations).

tau REAL for sgeqpf

DOUBLE PRECISION for dgeqpf

COMPLEX for cgeqpf

DOUBLE COMPLEX for zgeqpf.

Array, size at least max (1, min(*m*, *n*)). Contains additional information on the matrix *Q*.

jpvt Overwritten by details of the permutation matrix *P* in the factorization *A***P* = *Q***R*. More precisely, the columns of *A***P* are the columns of *A* in the following order:

jpvt(1), *jpvt*(2), ..., *jpvt*(*n*).

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine geqpf interface are the following:

a Holds the matrix *A* of size (*m*,*n*).

jpvt Holds the vector of length *n*.

tau Holds the vector of length min(*m*,*n*)

Application Notes

The computed factorization is the exact factorization of a matrix *A* + *E*, where

$||E||_2 = O(\varepsilon)||A||_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$	if $m = n$,
$(2/3)n^2(3m-n)$	if $m > n$,
$(2/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $||A*x - b||_2$ for all columns b of a given matrix B, you can call the following:

?geqpf (this routine)	to factorize $A*P = Q*R$;
ormqr	to compute $C = Q^T*B$ (for real matrices);
unmqr	to compute $C = Q^H*B$ (for complex matrices);
trsm (a BLAS routine)	to solve $R*X = C$.

(The columns of the computed X are the permuted least squares solution vectors x; the output array *jpvt* specifies the permutation order.)

To compute the elements of Q explicitly, call

orgqr	(for real matrices)
ungqr	(for complex matrices).

?geqp3

Computes the QR factorization of a general m-by-n matrix with column pivoting using level 3 BLAS.

Syntax

```
call sgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call dgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call cgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
call zgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
call geqp3(a, jpvt [,tau] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the *QR* factorization of a general *m*-by-*n* matrix *A* with column pivoting: $A*P = Q*R$ (see Orthogonal Factorizations) using Level 3 BLAS. Here *P* denotes an *n*-by-*n* permutation matrix. Use this routine instead of geqpf for better performance.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in *A* (*n*≥ 0).
a, *work*	REAL for sgeqp3 DOUBLE PRECISION for dgeqp3 COMPLEX for cgeqp3 DOUBLE COMPLEX for zgeqp3. Arrays: *a* (*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; must be at least max(1, 3*n+1) for real flavors, and at least max(1, *n*+1) for complex flavors. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* below for details.
jpvt	INTEGER. Array, size at least max(1, *n*). On entry, if *jpvt*(*i*)≠ 0, the *i*-th column of *A* is moved to the beginning of *AP* before the computation, and fixed in place during the computation. If *jpvt*(*i*) = 0, the *i*-th column of *A* is a free column (that is, it may be interchanged during the computation with any other free column).
rwork	REAL for cgeqp3 DOUBLE PRECISION for zgeqp3. A workspace array, size at least max(1, 2**n*). Used in complex flavors only.

Output Parameters

a	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the min(*m*,*n*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m*≥*n*); the elements below the diagonal, with the array *tau*, present the orthogonal matrix *Q* as a product of min(*m*,*n*) elementary reflectors (see Orthogonal Factorizations).
tau	REAL for sgeqp3 DOUBLE PRECISION for dgeqp3 COMPLEX for cgeqp3 DOUBLE COMPLEX for zgeqp3.

	Array, size at least max (1, min(*m*, *n*)). Contains scalar factors of the elementary reflectors for the matrix *Q*.
jpvt	Overwritten by details of the permutation matrix *P* in the factorization *A*P = Q*R*. More precisely, the columns of *AP* are the columns of *A* in the following order: *jpvt*(1), *jpvt*(2), ..., *jpvt*(*n*).
info	INTEGER. If *info = 0*, the execution is successful. If *info = -i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine geqp3 interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
jpvt	Holds the vector of length *n*.
tau	Holds the vector of length min(*m*,*n*)

Application Notes

To solve a set of least squares problems minimizing $||A*x - b||_2$ for all columns *b* of a given matrix *B*, you can call the following:

?geqp3 (this routine)	to factorize *A*P = Q*R*;
ormqr	to compute $C = Q^T*B$ (for real matrices);
unmqr	to compute $C = Q^H*B$ (for complex matrices);
trsm (a BLAS routine)	to solve *R*X = C*.

(The columns of the computed *X* are the permuted least squares solution vectors *x*; the output array *jpvt* specifies the permutation order.)

To compute the elements of *Q* explicitly, call

orgqr	(for real matrices)
ungqr	(for complex matrices).

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork = -1*.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork = -1*, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?orgqr
Generates the real orthogonal matrix Q of the QR factorization formed by `?geqrf`.

Syntax

```
call sorgqr(m, n, k, a, lda, tau, work, lwork, info)
call dorgqr(m, n, k, a, lda, tau, work, lwork, info)
call orgqr(a, tau [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine generates the whole or part of *m*-by-*m* orthogonal matrix *Q* of the *QR* factorization formed by the routine `?geqrf` or geqpf. Use this routine after a call to `sgeqrf`/`dgeqrf` or `sgeqpf`/`dgeqpf`.

Usually *Q* is determined from the *QR* factorization of an *m* by *p* matrix *A* with $m \geq p$. To compute the whole matrix *Q*, use:

```
call?orgqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading *p* columns of *Q* (which form an orthonormal basis in the space spanned by the columns of *A*):

```
call?orgqr(m, p, p, a, lda, work, lwork, info)
```

To compute the matrix Q^k of the *QR* factorization of leading *k* columns of the matrix *A*:

```
call?orgqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading *k* columns of Q^k (which form an orthonormal basis in the space spanned by leading *k* columns of the matrix *A*):

```
call?orgqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	`INTEGER`. The order of the orthogonal matrix *Q* ($m \geq 0$).
n	`INTEGER`. The number of columns of *Q* to be computed ($0 \leq n \leq m$).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* ($0 \leq k \leq n$).

a, *tau*, *work*	`REAL` for `sorgqr`
	`DOUBLE PRECISION` for `dorgqr`
	Arrays:
	a(*lda*,*) and *tau*(*) are the arrays returned by `sgeqrf` / `dgeqrf` or `sgeqpf` / `dgeqpf`.

The second dimension of *a* must be at least max(1, *n*).

The size of *tau* must be at least max(1, *k*).

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of *a*; at least max(1, *m*).

lwork — `INTEGER`. The size of the *work* array (*lwork*≥*n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by *n* leading columns of the *m*-by-*m* orthogonal matrix *Q*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `orgqr` interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

tau — Holds the vector of length (*k*)

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly orthogonal matrix by a matrix *E* such that

$||E||_2 = O(\varepsilon)||A||_2$ where ε is the machine precision.

The total number of floating-point operations is approximately $4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.

If $n = k$, the number is approximately $(2/3)*n^2*(3m - n)$.

The complex counterpart of this routine is ungqr.

?ormqr
Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by ?geqrf *or* ?geqpf.

Syntax

```
call sormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call ormqr(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a real matrix *C* by *Q* or Q^T, where *Q* is the orthogonal matrix *Q* of the *QR* factorization formed by the routine ?geqrf or ?geqpf.

Depending on the parameters *side*left_right and trans, the routine can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (overwriting the result on *C*).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side*='L', *Q* or Q^T is applied to *C* from the left. If *side*='R', *Q* or Q^T is applied to *C* from the right.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans*='N', the routine multiplies *C* by *Q*. If *trans*='T', the routine multiplies *C* by Q^T.
m	INTEGER. The number of rows in the matrix *C* ($m \geq 0$).
n	INTEGER. The number of columns in *C* ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints: $0 \leq k \leq m$ if *side*='L'; $0 \leq k \leq n$ if *side*='R'.
a, *tau*, *c*, *work*	REAL for sgeqrf DOUBLE PRECISION for dgeqrf. Arrays:

a(*lda*,*) and *tau*(*) are the arrays returned by `sgeqrf` / `dgeqrf` or `sgeqpf` / `dgeqpf`.

The second dimension of *a* must be at least max(1, *k*).

The size of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the matrix *C*.

The second dimension of *c* must be at least max(1, *n*)

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of *a*. Constraints:

if `side = 'L'`, *lda* ≥ `max(1, m)` ;

if `side = 'R'`, *lda* ≥ `max(1, n)`.

ldc — `INTEGER`. The leading dimension of *c*. Constraint:

ldc ≥ max(1, *m*).

lwork — `INTEGER`. The size of the *work* array. Constraints:

lwork ≥ `max(1, n)` if `side = 'L'`;

lwork ≥ `max(1, m)` if `side = 'R'`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).

work(1) — If `info = 0`, on exit `work(1)` contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `ormqr` interface are the following:

a — Holds the matrix *A* of size (*r*,*k*).

`r = m` if `side = 'L'`.

`r = n` if `side = 'R'`.

tau — Holds the vector of length (*k*).

c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork = n*blocksize* (if *side = 'L'*) or *lwork = m*blocksize* (if *side = 'R'*) where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork = -1*.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork = -1*, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is unmqr.

?gemqr
Multiples a matrix C by a real orthogonal or complex unitary matrix Q, as computed by ?geqr, with best performance for tall and skinny matrices.

```
call sgemqr(side, trans, m, n, k, a, lda, t, tsize, c, ldc, work, lwork, info)
call dgemqr(side, trans, m, n, k, a, lda, t, tsize, c, ldc, work, lwork, info)
call cgemqr(side, trans, m, n, k, a, lda, t, tsize, c, ldc, work, lwork, info)
call zgemqr(side, trans, m, n, k, a, lda, t, tsize, c, ldc, work, lwork, info)
```

Description

The ?gemqr routine multiplies an *m*-by-*n* matrix *C* by Op(*Q*), where matrix *Q* is the factor from the LQ factorization of matrix *A* formed by ?geqr, and

Op(*Q*) = *Q*, or

Op(*Q*) = Q^T, or

Op(*Q*) = Q^H.

NOTE
You must use ?geqr for LQ factorization before calling ?gemqr. ?gemqr is not compatible with QR factorization routines other than ?geqr.

For real flavors, *C* is real and *Q* is real orthogonal.

For complex flavors, *C* is complex and *Q* is complex unitary.

If matrix *A* is tall and skinny, a highly scalable algorithm is used to avoid communication overhead. Otherwise, ?ormqr or ?unmqr is used.

Input Parameters

side
CHARACTER*1.
If *side* = 'L': apply Op(*Q*) from the left.
If *side* = 'R'': apply Op(*Q*) from the right.

trans
CHARACTER*1.
If *trans* = 'N': No transpose, Op(*Q*) = *Q*.
If *trans* = 'T': Transpose, Op(*Q*) = Q^T.
If *trans* = 'C': Conjugate transpose, Op(*Q*) = Q^H.

m
INTEGER. The number of rows of the matrix *A*. *m*≥ 0.

n
INTEGER. The number of columns of the matrix *C*. *m*≥*n*≥ 0.

k
INTEGER. The number of elementary reflectors whose product defines the matrix Q.
If *side* = 'L', *m*≥*k*≥ 0.
if *side* = 'R', *n*≥*k*≥ 0.

a
REAL for sgemqr
DOUBLE PRECISION for dgemqr
COMPLEX for cgemqr
COMPLEX*16 for zgemqr
Array, size (*lda*,*k*).
Part of the data structure to represent *Q* as returned by ?geqr.

lda
INTEGER. The leading dimension of the array *a*.
If *side* = 'L', *lda*≥max(1,*m*).
If *side* = 'R', *lda*≥max(1,*n*).

t
REAL for sgemlq
DOUBLE PRECISION for dgemlq
COMPLEX for cgemlq
COMPLEX*16 for zgemlq
Array, size (max(5,*tsize*)). Part of the data structure to represent *Q* as returned by ?geqr.

tsize
INTEGER. The size of the array *t*. *tsize*≥ 5.

c
REAL for sgemqr
DOUBLE PRECISION for dgemqr
COMPLEX for cgemqr
COMPLEX*16 for zgemqr
Array of size (*ldc*,*n*). On entry, contains the *m*-by-*n* matrix *C*.

ldc	INTEGER. The leading dimension of the array *c*. *ldc*≥max(1,*m*).
lwork	INTEGER. The size of the array *work*. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array and returns this value as *work*(1); no error message related to *lwork* is issued by xerbla.

Output Parameters

c	On exit, *c* is overwritten by Op(*Q*)***C* or *C**Op(*Q*).
work	REAL for sgemlq DOUBLE PRECISION for dgemlq COMPLEX for cgemlq COMPLEX*16 for zgemlq Workspace array of size (max(1,*lwork*)).
info	INTEGER. *info* = 0 indicates a successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?ungqr
Generates the complex unitary matrix Q of the QR factorization formed by ?geqrf.

Syntax

```
call cungqr(m, n, k, a, lda, tau, work, lwork, info)
call zungqr(m, n, k, a, lda, tau, work, lwork, info)
call ungqr(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates the whole or part of *m*-by-*m* unitary matrix *Q* of the *QR* factorization formed by the routines ?geqrf or geqpf. Use this routine after a call to cgeqrf/zgeqrf or cgeqpf/zgeqpf.

Usually *Q* is determined from the *QR* factorization of an *m* by *p* matrix *A* with *m*≥*p*. To compute the whole matrix *Q*, use:

```
call?ungqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading *p* columns of *Q* (which form an orthonormal basis in the space spanned by the columns of *A*):

```
call?ungqr(m, p, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the *QR* factorization of the leading *k* columns of the matrix *A*:

```
call?ungqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading *k* columns of Q^k (which form an orthonormal basis in the space spanned by the leading *k* columns of the matrix *A*):

```
call?ungqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The order of the unitary matrix *Q* (*m*≥ 0).
n	INTEGER. The number of columns of *Q* to be computed (0 ≤*n*≤*m*).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (0 ≤*k*≤*n*).
a, *tau*, *work*	COMPLEX for cungqr DOUBLE COMPLEX for zungqr Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by cgeqrf/zgeqrf or cgeqpf/zgeqpf. The second dimension of *a* must be at least max(1, *n*). The size of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array (*lwork*≥*n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ungqr interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).

tau Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly unitary matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)*||A||_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $16*m*n*k - 8*(m + n)*k2 + (16/3)*k^3$.

If $n = k$, the number is approximately $(8/3)*n^2*(3m - n)$.

The real counterpart of this routine is orgqr.

?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by ?geqrf.

Syntax

```
call cunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call unmqr(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a rectangular complex matrix *C* by *Q* or Q^H, where *Q* is the unitary matrix *Q* of the *QR* factorization formed by the routines ?geqrf or geqpf.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products *Q*C*, Q^H*C*, *C*Q*, or *C*Q^H (overwriting the result on *C*).

Input Parameters

side CHARACTER*1. Must be either 'L' or 'R'.

If *side* = 'L', *Q* or Q^H is applied to *C* from the left.

If *side* = 'R', *Q* or Q^H is applied to *C* from the right.

trans — CHARACTER*1. Must be either 'N' or 'C'.

If *trans* = 'N', the routine multiplies *C* by *Q*.

If *trans* = 'C', the routine multiplies *C* by Q^H.

m — INTEGER. The number of rows in the matrix *C* (*m* ≥ 0).

n — INTEGER. The number of columns in *C* (*n* ≥ 0).

k — INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

0 ≤ *k* ≤ *m* if *side* = 'L';

0 ≤ *k* ≤ *n* if *side* = 'R'.

a, *c*, *tau*, *work* — COMPLEX for cgeqrf

DOUBLE COMPLEX for zgeqrf.

Arrays:

a(*lda*,*) and *tau*(*) are the arrays returned by cgeqrf / zgeqrf or cgeqpf / zgeqpf.

The second dimension of *a* must be at least max(1, *k*).

The size of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.

The second dimension of *c* must be at least max(1, *n*)

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*. Constraints:

lda ≥ max(1, *m*) if *side* = 'L';

lda ≥ max(1, *n*) if *side* = 'R'.

ldc — INTEGER. The leading dimension of *c*. Constraint:

ldc ≥ max(1, *m*) .

lwork — INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).

work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine unmqr interface are the following:

a	Holds the matrix *A* of size (*r*,*k*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ormqr.

?gelqf
Computes the LQ factorization of a general m-by-n matrix.

Syntax

```
call sgelqf(m, n, a, lda, tau, work, lwork, info)
call dgelqf(m, n, a, lda, tau, work, lwork, info)
call cgelqf(m, n, a, lda, tau, work, lwork, info)
```

```
call zgelqf(m, n, a, lda, tau, work, lwork, info)
call gelqf(a [, tau] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the *LQ* factorization of a general *m*-by-*n* matrix *A* (see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in *A* (*n*≥ 0).
a, *work*	REAL for sgelqf DOUBLE PRECISION for dgelqf COMPLEX for cgelqf DOUBLE COMPLEX for zgelqf. Arrays: Array *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *m*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the factorization data as follows: The elements on and below the diagonal of the array contain the *m*-by-min(*m*,*n*) lower trapezoidal matrix *L* (*L* is lower triangular if *m*≤*n*); the elements above the diagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors.

tau	REAL for sgelqf DOUBLE PRECISION for dgelqf COMPLEX for cgelqf DOUBLE COMPLEX for zgelqf. Array, size at least max(1, min(*m*, *n*)). Contains scalars that define elementary reflectors for the matrix *Q* (see Orthogonal Factorizations).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gelqf interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length min(*m*,*n*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$||E||_2 = O(\varepsilon) \ ||A||_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$	if $m = n$,
$(2/3)n^2(3m-n)$	if $m > n$,
$(2/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

To find the minimum-norm solution of an underdetermined least squares problem minimizing $||A*x - b||_2$ for all columns *b* of a given matrix *B*, you can call the following:

?gelqf (this routine)	to factorize $A = L*Q$;
trsm (a BLAS routine)	to solve $L*Y = B$ for *Y*;
ormlq	to compute $X = (Q_1)^T*Y$ (for real matrices);
unmlq	to compute $X = (Q_1)^H*Y$ (for complex matrices).

(The columns of the computed *X* are the minimum-norm solution vectors *x*. Here *A* is an *m*-by-*n* matrix with $m < n$; Q_1 denotes the first *m* columns of *Q*).

To compute the elements of *Q* explicitly, call

orglq	(for real matrices)
unglq	(for complex matrices).

See Also

mkl_progress

Matrix Storage Schemes

?gelq
Computes an LQ factorization of a general matrix.

```
call sgelq(m, n, a, lda, t, tsize, work, lwork, info)
call dgelq(m, n, a, lda, t, tsize, work, lwork, info)
call cgelq(m, n, a, lda, t, tsize, work, lwork, info)
call zgelq(m, n, a, lda, t, tsize, work, lwork, info)
```

Description

The ?gelq routines computes an LQ factorization of an *m*-by-*n* matrix *A*. If the matrix is short and wide (*n* is substantially larger than *m*), a highly scalable algorithm is used to avoid communication overhead.

NOTE
The internal format of the elementary reflectors generated by ?gelq is only compatible with the ?gemlq routine and not any other LQ routines.

NOTE
An optimized version of ?gelq is not available.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. $m \geq 0$.
n	INTEGER. The number of columns of the matrix *A*. $n \geq 0$.
a	REAL for sgelq

DOUBLE PRECISION for dgelq

COMPLEX for cgelq

COMPLEX*16 for zgelq

Array, size (*lda*, *n*). Contains the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*. *lda*≥max(1, *m*).

tsize INTEGER. If *tsize*≥ 5, the size of the array *t*. If *tsize* = -1 or *tsize* = -2, then the routine performs a workspace query. The routine calculates the sizes required for the *t* and *work* arrays and returns these values as the first entries of the *t* and *work* arrays, without issuing any error message related to *t* or *work* by xerbla.

If *tsize* = -1, the routine calculates the optimal size of *t* for optimum performance and returns this value in *t*(1).

If *tsize* = -2, the routine calculates then minimum size required for *t* and returns this value in *t*(1).

lwork INTEGER. The size of the array *work*. If *lwork* = -1 or *lwork* = -2, then the routine performs a workspace query. The routine only calculates the sizes of the *t* and *work* arrays and returns these values as the first entries of the *t* and *work* arrays, without issuing any error message related to *t* or *work* by xerbla.

If *lwork* = -1, the routine calculates the optimal size of *work* for optimum performance and returns this value in *work*(1).

If *lwork* = -2, the routine calculates the minimum size required for *work* and returns this value in *work*(1).

Output Parameters

a REAL for sgelq

DOUBLE PRECISION for dgelq

COMPLEX for cgelq

COMPLEX*16 for zgelq

The elements on and below the diagonal of the array contain the lower trapezoidal matrix *L* (*L* is lower triangular if *m*≤*n*). The elements above the diagonal are used to store part of the internal data structure representing *Q*.

t REAL for sgelq

DOUBLE PRECISION for dgelq

COMPLEX for cgelq

COMPLEX*16 for zgelq

Array, size (max(5,*tsize*)).

If *info* = 0, *t*(1) returns the optimal value for *tsize*. You can specify that it return the minimum required value for *tsize* instead - see the *tsize* description for details. The remaining entries of *t* contains part of the data structure used to represent *Q*. To apply or construct Q, you need to retain *a* and *t* and pass them to other routines.

work

REAL for sgelq

DOUBLE PRECISION for dgelq

COMPLEX for cgelq

COMPLEX*16 for zgelq

Array, size (max(1,*lwork*)).

If *info* = 0, *work*(1) contains the optimal value for *lwork*. You can specify that it return the minimum required value for *lwork* instead - see the *lwork* description for details.

info

INTEGER.

info = 0 indicates a successful exit.

info < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

See Also

?gemlq Multiples a matrix *C* by a real orthogonal or complex unitary matrix *Q*, as computed by ?gelq.

?gelqt

?gelqt *computes a blocked LQ factorization of a real or complex m-by-n matrix A using the compact WY representation of Q.*

```
call sgelqt(m, n, mb, a, lda, t, ldt, work, info)
call dgelqt(m, n, mb, a, lda, t, ldt, work, info)
call cgelqt(m, n, mb, a, lda, t, ldt, work, info)
call zgelqt(m, n, mb, a, lda, t, ldt, work, info)
```

Description

?gelqt computes a blocked LQ factorization of a real or complex *m*-by-*n* matrix *A* using the compact WY representation of *Q*.

The matrix *V* stores the elementary reflectors *H*(*i*) in the *i*-th row above the diagonal. For example, if *m*=5 and *n*=3, the matrix *V* is

$$V = \begin{pmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{pmatrix}$$

where the v_is represent the vectors which define *H*(*i*), which are returned in the array *a*. The 1 elements along the diagonal of *V* are not stored in *a*. Let *k* = min(*m*,*n*). The number of blocks is *b* = ceiling(*k*/*mb*), where each block is of order *mb* except for the last block, which is of order *ib* = *k* - (*b*-1)**mb*. For each of the *b* blocks, a upper triangular block reflector factor is computed: *T1*, *T2*, ..., *TB*. The *mb*-by-*mb* (and *ib*-by-*ib* for the last block) *T*'s are stored in the *mb*-by-*k* matrix *T* as

T = (*T1T2* ... *TB*).

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. *m*≥ 0.
n	INTEGER. The number of columns of the matrix *A*. *n*≥ 0.
mb	INTEGER. The block size to be used in the blocked QR. min(*m*, *n*) ≥*mb*≥ 1.
a	REAL for sgelqt DOUBLE PRECISION for dgelqt COMPLEX for cgelqt COMPLEX*16 for zgelqt Array of size (*lda*,*n*). On entry, the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*m*).
ldt	INTEGER. The leading dimension of the array *t*. *ldt*≥*mb*.

Output Parameters

a	On exit, the elements on and below the diagonal of the array contain the *m*-by-min(*m*, *n*) lower trapezoidal matrix *L* (*L* is lower triangular if *m*≤*n*); the elements above the diagonal are the rows of *V*.
t	REAL for sgelqt DOUBLE PRECISION for dgelqt COMPLEX for cgelqt COMPLEX*16 for zgelqt Array of size (*ldt*, min(*m*, *n*)). The upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks. See Description for further details.
work	REAL for sgelqt DOUBLE PRECISION for dgelqt COMPLEX for cgelqt COMPLEX*16 for zgelqt Array of size (*mb***n*).
info	INTEGER. *info* = 0: successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?gemlqt

Multiplies a general matrix by the orthogonal/unitary matrix Q of the LQ factorization formed by ?gelqt.

```
call sgemlqt(side, trans, m, n, k, mb, v, ldv, t, ldt, c, ldc, work, info)
call dgemlqt(side, trans, m, n, k, mb, v, ldv, t, ldt, c, ldc, work, info)
call cgemlqt(side, trans, m, n, k, mb, v, ldv, t, ldt, c, ldc, work, info)
```

```
call zgemlqt(side, trans, m, n, k, mb, v, ldv, t, ldt, c, ldc, work, info)
```

Description

?gemlqt overwrites the general real *m*-by-*n* matrix *C* with

	side = 'L'	*side* = 'R'
trans = 'N'	*Q***C*	*C***Q*
trans = 'T'	Q^T**C*	*C**Q^T
trans = 'C'	Q^H**C*	*C**Q^H

where *Q* is a real or complex orthogonal matrix defined as the product of *k* elementary reflectors:

Q = *H*(*k*) *H*(*k* - 1) . . . *H*(1) = *I* - *VTV*^{T} for real flavors

or

Q = *H*(*k*)^H*H*(*k* - 1)^H . . . *H*(1)^H = *I* - *VTV*^{H} for complex flavors

generated using the compact WY representation as returned by ?gelqt.

Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side
CHARACTER*1.
If *side*= 'L': apply op(*Q*) from the left;
if *side*= 'R': apply op(*Q*) from the right.

trans
CHARACTER*1.
If *trans* = 'N': No transpose, op(*Q*) = *Q*;
if *trans* = 'T': Transpose, op(*Q*) = Q^T;
if *trans*= 'C': Transpose, op(*Q*) = Q^H.

m
INTEGER. The number of rows of the matrix *C*. *m*≥ 0.

n
INTEGER. The number of columns of the matrix *C*. *n*≥ 0.

k
INTEGER. The number of elementary reflectors whose product defines the matrix *Q*.
If *side* = 'L', *m*≥*k*≥ 0;
if *side* = 'R', *n*≥*k*≥ 0.

mb
INTEGER. The block size used for the storage of *T*. *k*≥*mb*≥ 1. This must be the same value of *mb* used to generate *T* in ?gelqt.

v
REAL for sgemlqt
DOUBLE PRECISION for dgemlqt
COMPLEX for cgemlqt
COMPLEX*16 for zgemlqt

	Array of size (*ldv*,*m*) if *side* = 'L' or (*ldv*,*n*) if *side* = 'R'. The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,...,*k*, as returned by ?gelqt in the first *k* rows of its array argument *a*.
ldv	INTEGER. The leading dimension of the array *v*. *ldv*≥ max(1,*k*).
t	REAL for sgemlqt DOUBLE PRECISION for dgemlqt COMPLEX for cgemlqt COMPLEX*16 for zgemlqt Array of size (*ldt*,*k*). The upper triangular factors of the block reflectors as returned by ?gelqt, stored as a *mb*-by-*k* matrix.
ldt	INTEGER. The leading dimension of the array *t*. *ldt*≥*mb*.
c	REAL for sgemlqt DOUBLE PRECISION for dgemlqt COMPLEX for cgemlqt COMPLEX*16 for zgemlqt Array of size (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.
ldc	INTEGER. The leading dimension of the array *c*. *ldc*≥ max(1,*m*).

Output Parameters

c	On exit, *c* is overwritten by op(*Q*)**C* or *C**op(*Q*).
work	REAL for sgemlqt DOUBLE PRECISION for dgemlqt COMPLEX for cgemlqt COMPLEX*16 for zgemlqt Array. The size of *work* is *n***mb* if *side* = 'L', or *m***mb* if *side* = 'R'.
info	*info* = 0: successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?orglq
Generates the real orthogonal matrix Q of the LQ factorization formed by ?gelqf*.*

Syntax

```
call sorglq(m, n, k, a, lda, tau, work, lwork, info)
call dorglq(m, n, k, a, lda, tau, work, lwork, info)
call orglq(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates the whole or part of *n*-by-*n* orthogonal matrix *Q* of the *LQ* factorization formed by the routines gelqf. Use this routine after a call to sgelqf/dgelqf.

Usually *Q* is determined from the *LQ* factorization of an *p*-by-*n* matrix *A* with $n \geq p$. To compute the whole matrix *Q*, use:

```
call ?orglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading *p* rows of *Q*, which form an orthonormal basis in the space spanned by the rows of *A*, use:

```
call ?orglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the *LQ* factorization of the leading *k* rows of *A*, use:

```
call ?orglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading *k* rows of Q^k, which form an orthonormal basis in the space spanned by the leading *k* rows of *A*, use:

```
call ?orgqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m

INTEGER. The number of rows of *Q* to be computed

($0 \leq m \leq n$).

n

INTEGER. The order of the orthogonal matrix *Q* ($n \geq m$).

k

INTEGER. The number of elementary reflectors whose product defines the matrix *Q* ($0 \leq k \leq m$).

a, *tau*, *work*

REAL for sorglq

DOUBLE PRECISION for dorglq

Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by sgelqf/dgelqf.

The second dimension of *a* must be at least max(1, *n*).

The size of *tau* must be at least max(1, *k*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of *a*; at least max(1, *m*).

lwork

INTEGER. The size of the *work* array; at least max(1, *m*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by *m* leading rows of the *n*-by-*n* orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine orglq interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)*||A||_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.

If $m = k$, the number is approximately $(2/3)*m^2*(3n - m)$.

The complex counterpart of this routine is unglq.

?ormlq

Multiplies a real matrix by the orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```
call sormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call ormlq(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a real *m*-by-*n* matrix *C* by *Q* or Q^T, where *Q* is the orthogonal matrix *Q* of the *LQ* factorization formed by the routine gelqf.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (overwriting the result on *C*).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', *Q* or Q^T is applied to *C* from the left. If *side* = 'R', *Q* or Q^T is applied to *C* from the right.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies *C* by *Q*. If *trans* = 'T', the routine multiplies *C* by Q^T.
m	INTEGER. The number of rows in the matrix *C* ($m \geq 0$).
n	INTEGER. The number of columns in *C* ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints: $0 \leq k \leq m$ if *side* = 'L'; $0 \leq k \leq n$ if *side* = 'R'.
a, *c*, *tau*, *work*	REAL for sormlq DOUBLE PRECISION for dormlq. Arrays: *a*(*lda*,*) and *tau*(*) are arrays returned by ?gelqf. The second dimension of *a* must be: at least max(1, *m*) if *side* = 'L'; at least max(1, *n*) if *side* = 'R'. The dimension of *tau* must be at least max(1, *k*). *c*(*ldc*,*) contains the *m*-by-*n* matrix *C*. The second dimension of *c* must be at least max(1, *n*) *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *k*).
ldc	INTEGER. The leading dimension of *c*; *ldc* ≥ max(1, *m*).
lwork	INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ormlq interface are the following:

a — Holds the matrix *A* of size (*k*,*m*).

tau — Holds the vector of length (*k*).

c — Holds the matrix *C* of size (*m*,*n*).

side — Must be 'L' or 'R'. The default value is 'L'.

trans — Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork*= -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork*= -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is unmlq.

?gemlq
Multiples a matrix C by a real orthogonal or complex unitary matrix Q, as computed by ?gelq*.*

```
call sgemlq(side, trans, m, n, k, a, lda, t, tsize, c, ldc, work, lwork, info)
call dgemlq(side, trans, m, n, k, a, lda, t, tsize, c, ldc, work, lwork, info)
call cgemlq(side, trans, m, n, k, a, lda, t, tsize, c, ldc, work, lwork, info)
call zgemlq(side, trans, m, n, k, a, lda, t, tsize, c, ldc, work, lwork, info)
```

Description

The ?gemlq routine multiplies an *m*-by-*n* matrix *C* by Op(*Q*), where matrix *Q* is the factor from the LQ factorization of matrix *A* formed by ?gelq, and

Op(*Q*) = *Q*, or

Op(*Q*) = Q^T, or

Op(*Q*) = Q^H.

NOTE
You must use ?gelq for LQ factorization before calling ?gemlq. ?gemlq is not compatible with LQ factorization routines other than ?gelq.

For real flavors, *C* is real and *Q* is real orthogonal.

For complex flavors, *C* is complex and *Q* is complex unitary.

If matrix *A* is short and wide, a highly scalable algorithm is used to avoid communication overhead. Otherwise, ?ormlq or ?unmlq is used.

NOTE
An optimized version of ?gemlq is not available.

Input Parameters

side
CHARACTER*1.
If *side* = 'L': apply Op(*Q*) from the left.
If *side* = 'R': apply Op(*Q*) from the right.

trans
CHARACTER*1.
If *trans* = 'N': No transpose, Op(*Q*) = *Q*.
If *trans* = 'T': Transpose, Op(*Q*) = Q^T.
If *trans* = 'C': Conjugate transpose, Op(*Q*) = Q^H.

m
INTEGER. The number of rows of the matrix *A*. $m \geq 0$.

n
INTEGER. The number of columns of the matrix *C*. $n \geq 0$.

k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. If *side* = 'L', $m \geq k \geq 0$. if *side* = 'R', $n \geq k \geq 0$.
a	REAL for sgemlq DOUBLE PRECISION for dgemlq COMPLEX for cgemlq COMPLEX*16 for zgemlq Array of size (*lda*,*m*) if *side* = 'L', or (*lda*,*n*) if *side* = 'R'. Part of the data structure to represent *Q* as returned by ?gelq.
lda	INTEGER . The leading dimension of the array *a*. $lda \geq \max(1,k)$.
t	REAL for sgemlq DOUBLE PRECISION for dgemlq COMPLEX for cgemlq COMPLEX*16 for zgemlq Array, size (max(5,*tsize*)). Part of the data structure to represent *Q* as returned by ?gelq.
tsize	INTEGER. The size of the array *t*. $tsize \geq 5$.
c	REAL for sgemlq DOUBLE PRECISION for dgemlq COMPLEX for cgemlq COMPLEX*16 for zgemlq Array, size (*ldc*,*n*). On entry, *a* contains the *m*-by-*n* matrix *C*.
ldc	INTEGER. The leading dimension of the array *c*. $ldc \geq \max(1,m)$.
lwork	INTEGER. The size of the array *work*. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array and returns this value as *work*(1); no error message related to *lwork* is issued by xerbla.

Output Parameters

c	On exit, *c* is overwritten by Op(*Q*)***C* or *C**Op(*Q*).
work	REAL for sgemlq DOUBLE PRECISION for dgemlq COMPLEX for cgemlq COMPLEX*16 for zgemlq Workspace array of size (max(1,*lwork*)).

info	INTEGER. *info* = 0 indicates a successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?unglq

Generates the complex unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```
call cunglq(m, n, k, a, lda, tau, work, lwork, info)
call zunglq(m, n, k, a, lda, tau, work, lwork, info)
call unglq(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates the whole or part of *n*-by-*n* unitary matrix *Q* of the *LQ* factorization formed by the routines gelqf. Use this routine after a call to cgelqf/zgelqf.

Usually *Q* is determined from the *LQ* factorization of an *p*-by-*n* matrix *A* with $n < p$. To compute the whole matrix *Q*, use:

```
call ?unglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading *p* rows of *Q*, which form an orthonormal basis in the space spanned by the rows of *A*, use:

```
call ?unglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the *LQ* factorization of the leading *k* rows of *A*, use:

```
call ?unglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading *k* rows of Q^k, which form an orthonormal basis in the space spanned by the leading *k* rows of *A*, use:

```
call ?ungqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The number of rows of *Q* to be computed ($0 \le m \le n$).
n	INTEGER. The order of the unitary matrix *Q* ($n \ge m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* ($0 \le k \le m$).
a, *tau*, *work*	COMPLEX for cunglq DOUBLE COMPLEX for zunglq Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by cgelqf/zgelqf. The second dimension of *a* must be at least max(1, *n*).

The dimension of *tau* must be at least max(1, *k*).

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of *a*; at least max(1, *m*).

lwork — `INTEGER`. The size of the *work* array; at least max(1, *m*).

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by *m* leading rows of the *n*-by-*n* unitary matrix *Q*.

work(1) — If `info = 0`, on exit `work(1)` contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `unglq` interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

tau — Holds the vector of length (*k*).

Application Notes

For better performance, try using `lwork = m*blocksize`, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly unitary matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)*||A||_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $16*m*n*k - 8*(m + n)*k^2 + (16/3)*k^3$.

If $m = k$, the number is approximately $(8/3)*m^2*(3n - m)$.

The real counterpart of this routine is orglq.

?unmlq

Multiplies a complex matrix by the unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```
call cunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call unmlq(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a real *m*-by-*n* matrix *C* by *Q* or Q^H, where *Q* is the unitary matrix *Q* of the *LQ* factorization formed by the routine gelqf.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products *Q***C*, Q^H**C*, *C***Q*, or *C**Q^H (overwriting the result on *C*).

Input Parameters

side

CHARACTER*1. Must be either 'L' or 'R'.

If *side* = 'L', *Q* or Q^H is applied to *C* from the left.

If *side* = 'R', *Q* or Q^H is applied to *C* from the right.

trans

CHARACTER*1. Must be either 'N' or 'C'.

If *trans* = 'N', the routine multiplies *C* by *Q*.

If *trans* = 'C', the routine multiplies *C* by Q^H.

m

INTEGER. The number of rows in the matrix *C* ($m \geq 0$).

n

INTEGER. The number of columns in *C* ($n \geq 0$).

k

INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

$0 \leq k \leq m$ if *side* = 'L';

$0 \leq k \leq n$ if *side* = 'R'.

a, *c*, *tau*, *work*

COMPLEX for cunmlq

DOUBLE COMPLEX for zunmlq.

Arrays:

a(*lda*,*) and *tau*(*) are arrays returned by ?gelqf.

The second dimension of *a* must be:

at least max(1, *m*) if *side* = 'L';

at least max(1, *n*) if *side* = 'R'.

The size of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.

The second dimension of *c* must be at least max(1, *n*)

work is a workspace array, its dimension max(1, *lwork*).

lda INTEGER. The leading dimension of *a*; *lda*≥ max(1, *k*).

ldc INTEGER. The leading dimension of *c*; *ldc*≥ max(1, *m*).

lwork INTEGER. The size of the *work* array. Constraints:

lwork≥ max(1, *n*) if *side* = 'L';

lwork≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c Overwritten by the product $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine unmlq interface are the following:

a Holds the matrix *A* of size (*k*,*m*).

tau Holds the vector of length (*k*).

c Holds the matrix *C* of size (*m*,*n*).

side Must be 'L' or 'R'. The default value is 'L'.

trans Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ormlq.

?geqlf

Computes the QL factorization of a general m-by-n matrix.

Syntax

```
call sgeqlf(m, n, a, lda, tau, work, lwork, info)
call dgeqlf(m, n, a, lda, tau, work, lwork, info)
call cgeqlf(m, n, a, lda, tau, work, lwork, info)
call zgeqlf(m, n, a, lda, tau, work, lwork, info)
call geqlf(a [, tau] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine forms the *QL* factorization of a general *m*-by-*n* matrix *A* (see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

`m`	`INTEGER`. The number of rows in the matrix *A* (`m≥ 0`).
`n`	`INTEGER`. The number of columns in *A* (`n≥ 0`).

a, *work*

REAL for sgeqlf

DOUBLE PRECISION for dgeqlf

COMPLEX for cgeqlf

DOUBLE COMPLEX for zgeqlf.

Arrays:

Array *a*(*lda*,*) contains the matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of *a*; at least max(1, *m*).

lwork

INTEGER. The size of the *work* array; at least max(1, *n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a

Overwritten on exit by the factorization data as follows:

if $m \geq n$, the lower triangle of the subarray *a*(*m*-*n*+1:*m*, 1:*n*) contains the *n*-by-*n* lower triangular matrix *L*; if $m \leq n$, the elements on and below the (*n*-*m*)-th superdiagonal contain the *m*-by-*n* lower trapezoidal matrix *L*; in both cases, the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

tau

REAL for sgeqlf

DOUBLE PRECISION for dgeqlf

COMPLEX for cgeqlf

DOUBLE COMPLEX for zgeqlf.

Array, size at least max(1, min(*m*, *n*)). Contains scalar factors of the elementary reflectors for the matrix *Q* (see Orthogonal Factorizations).

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine geqlf interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length min(*m*,*n*).

Application Notes

For better performance, try using *lwork =n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

orgql	to generate matrix Q (for real matrices);
ungql	to generate matrix Q (for complex matrices);
ormql	to apply matrix Q (for real matrices);
unmql	to apply matrix Q (for complex matrices).

See Also

mkl_progress

Matrix Storage Schemes

?orgql
Generates the real matrix Q of the QL factorization formed by ?geqlf.

Syntax

```
call sorgql(m, n, k, a, lda, tau, work, lwork, info)
call dorgql(m, n, k, a, lda, tau, work, lwork, info)
call orgql(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates an *m*-by-*n* real matrix *Q* with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors *H*(*i*) of order *m*: *Q = H*(*k*) **...* H*(2)**H*(1) as returned by the routines geqlf. Use this routine after a call to sgeqlf/dgeqlf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q* (*m* ≥ 0).
n	INTEGER. The number of columns of the matrix *Q* (*m* ≥ *n* ≥ 0).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*n* ≥ *k* ≥ 0).
a, *tau*, *work*	REAL for sorgql DOUBLE PRECISION for dorgql Arrays: *a*(*lda*,*), *tau*(*). On entry, the (*n* - *k* + *i*)th column of *a* must *contain the vector which* defines the elementary reflector *H*(*i*), for *i* = 1,2,...,*k*, as returned by sgeqlf/dgeqlf in the last *k* columns of its array argument *a*; *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by sgeqlf/dgeqlf; The second dimension of *a* must be at least max(1, *n*). The size of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *n*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the last *n* columns of the *m*-by-*m* orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine orgql interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using `lwork =n*blocksize`, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is ungql.

?ungql

Generates the complex matrix Q of the QL factorization formed by `?geqlf`*.*

Syntax

```
call cungql(m, n, k, a, lda, tau, work, lwork, info)
call zungql(m, n, k, a, lda, tau, work, lwork, info)
call ungql(a, tau [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine generates an *m*-by-*n* complex matrix *Q* with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors `H(i)` of order *m*: `Q = H(k) *...* H(2)*H(1)` as returned by the routines geqlf/geqlf . Use this routine after a call to `cgeqlf`/`zgeqlf`.

Input Parameters

`m`	`INTEGER`. The number of rows of the matrix *Q* (`m≥0`).
`n`	`INTEGER`. The number of columns of the matrix *Q* (`m≥n≥0`).
`k`	`INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* (`n≥k≥0`).

`a`, `tau`, `work`	`COMPLEX` for `cungql`
	`DOUBLE COMPLEX` for `zungql`
	Arrays: *a*(*lda*,*), *tau*(*), *work*(*lwork*).
	On entry, the (*n* - *k* + *i*)th column of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by `cgeqlf`/`zgeqlf` in the last *k* columns of its array argument *a*;

tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by `cgeqlf`/`zgeqlf`;

The second dimension of *a* must be at least max(1, *n*).

The size of *tau* must be at least max(1, *k*).

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of *a*; at least max(1, *m*).

lwork — `INTEGER`. The size of the *work* array; at least max(1, *n*).

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the last *n* columns of the `m`-by-`m` unitary matrix *Q*.

work(1) — If `info = 0`, on exit `work(1)` contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `ungql` interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

tau — Holds the vector of length (*k*).

Application Notes

For better performance, try using `lwork =n*blocksize`, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is orgql.

?ormql

Multiplies a real matrix by the orthogonal matrix Q of the QL factorization formed by `?geqlf`.

Syntax

```
call sormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call ormql(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a real *m*-by-*n* matrix *C* by *Q* or Q^T, where *Q* is the orthogonal matrix *Q* of the *QL* factorization formed by the routine geqlf.

Depending on the parameters *side* and *trans*, the routine ormql can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (overwriting the result over *C*).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', *Q* or Q^T is applied to *C* from the left. If *side* = 'R', *Q* or Q^T is applied to *C* from the right.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies *C* by *Q*. If *trans* = 'T', the routine multiplies *C* by Q^T.
m	INTEGER. The number of rows in the matrix *C* ($m \geq 0$).
n	INTEGER. The number of columns in *C* ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints: $0 \leq k \leq m$ if *side* = 'L'; $0 \leq k \leq n$ if *side* = 'R'.
a, *tau*, *c*, *work*	REAL for sormql DOUBLE PRECISION for dormql. Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*). On entry, the *i*th column of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by sgeqlf/dgeqlf in the last *k* columns of its array argument *a*. The second dimension of *a* must be at least max(1, *k*).

tau(*i*) must contain the scalar factor of the elementary reflector H_i, as returned by `sgeqlf`/`dgeqlf`.

The size of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.

The second dimension of *c* must be at least max(1, *n*)

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of *a*;

if `side = 'L'`, *lda* ≥ `max(1, m)` ;

if `side = 'R'`, *lda* ≥ `max(1, n)`.

ldc — `INTEGER`. The leading dimension of *c*; *ldc* ≥ `max(1, m)`.

lwork — `INTEGER`. The size of the *work* array. Constraints:

lwork ≥ `max(1, n)` if `side = 'L'`;

lwork ≥ `max(1, m)` if `side = 'R'`.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `ormql` interface are the following:

a — Holds the matrix *A* of size (*r*,*k*).

r = *m* if `side = 'L'`.

r = *n* if `side = 'R'`.

tau — Holds the vector of length (*k*).

c — Holds the matrix *C* of size (*m*,*n*).

side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork = n*blocksize* (if *side = 'L'*) or *lwork = m*blocksize* (if *side = 'R'*) where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork = -1*.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork = -1*, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is unmql.

?unmql

Multiplies a complex matrix by the unitary matrix Q of the QL factorization formed by ?geqlf.

Syntax

```
call cunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call unmql(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a complex *m*-by-*n* matrix *C* by *Q* or Q^H, where *Q* is the unitary matrix *Q* of the *QL* factorization formed by the routine geqlf.

Depending on the parameters *side* and *trans*, the routine unmql can form one of the matrix products *Q*C*, Q^H**C*, *C*Q*, or *C**Q^H (overwriting the result over *C*).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side = 'L'*, *Q* or Q^H is applied to *C* from the left. If *side = 'R'*, *Q* or Q^H is applied to *C* from the right.
trans	CHARACTER*1. Must be either 'N' or 'C'. If *trans = 'N'*, the routine multiplies *C* by *Q*.

If *trans* = 'C', the routine multiplies *C* by Q^H.

m — INTEGER. The number of rows in the matrix *C* (*m* ≥ 0).

n — INTEGER. The number of columns in *C* (*n* ≥ 0).

k — INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

0 ≤ *k* ≤ *m* if *side* = 'L';

0 ≤ *k* ≤ *n* if *side* = 'R'.

a, *tau*, *c*, *work* — COMPLEX for cunmql

DOUBLE COMPLEX for zunmql.

Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).

On entry, the *i*-th column of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by cgeqlf/zgeqlf in the last *k* columns of its array argument *a*.

The second dimension of *a* must be at least max(1, *k*).

tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by cgeqlf/zgeqlf.

The size of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.

The second dimension of *c* must be at least max(1, *n*)

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *n*).

ldc — INTEGER. The leading dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product *Q***C*, Q^H**C*, *C***Q*, or *C**Q^H (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `unmql` interface are the following:

a	Holds the matrix *A* of size (*r*,*k*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'L'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = 'L') or *lwork* = *m*blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ormql.

?gerqf
Computes the RQ factorization of a general m-by-n matrix.

Syntax

```
call sgerqf(m, n, a, lda, tau, work, lwork, info)
call dgerqf(m, n, a, lda, tau, work, lwork, info)
call cgerqf(m, n, a, lda, tau, work, lwork, info)
call zgerqf(m, n, a, lda, tau, work, lwork, info)
call gerqf(a [, tau] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the *RQ* factorization of a general *m*-by-*n* matrix *A*(see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in *A* (*n*≥ 0).
a, *work*	REAL for sgerqf DOUBLE PRECISION for dgerqf COMPLEX for cgerqf DOUBLE COMPLEX for zgerqf. Arrays: Array *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; *lwork*≥ max(1, *m*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten on exit by the factorization data as follows: if *m*≤*n*, the upper triangle of the subarray *a*(1:*m*, *n*-*m*+1:*n*) contains the *m*-by-*m* upper triangular matrix *R*; if *m*≥*n*, the elements on and above the (*m*-*n*)th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*;

	in both cases, the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of min(*m*,*n*) elementary reflectors.

tau	`REAL` for `sgerqf`
	`DOUBLE PRECISION` for `dgerqf`
	`COMPLEX` for `cgerqf`
	`DOUBLE COMPLEX` for `zgerqf`.
	Array, size at least max (1, min(*m*, *n*)). (See Orthogonal Factorizations.)
	Contains scalar factors of the elementary reflectors for the matrix *Q*.

work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gerqf` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length min(*m*,*n*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

orgrq	to generate matrix Q (for real matrices);
ungrq	to generate matrix Q (for complex matrices);
ormrq	to apply matrix Q (for real matrices);

unmrq	to apply matrix Q (for complex matrices).

See Also

mkl_progress

Matrix Storage Schemes

?orgrq
Generates the real matrix Q of the RQ factorization formed by ?gerqf.

Syntax

```
call sorgrq(m, n, k, a, lda, tau, work, lwork, info)
call dorgrq(m, n, k, a, lda, tau, work, lwork, info)
call orgrq(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates an *m*-by-*n* real matrix with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors $H(i)$ of order *n*: $Q = H(1) * H(2) * \ldots * H(k)$ as returned by the routines gerqf. Use this routine after a call to sgerqf/dgerqf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q* ($m \geq 0$).
n	INTEGER. The number of columns of the matrix *Q* ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* ($m \geq k \geq 0$).
a, *tau*, *work*	REAL for sorgrq DOUBLE PRECISION for dorgrq Arrays: *a*(*lda*,*), *tau*(*). On entry, the (*m* - *k* + *i*)-th row of *a* must contain the vector which defines the elementary reflector $H(i)$, for i = 1,2,...,*k*, as returned by sgerqf/dgerqf in the last *k* rows of its array argument *a*; *tau*(i) must contain the scalar factor of the elementary reflector $H(i)$, as returned by sgerqf/dgerqf; The second dimension of *a* must be at least max(1, *n*). The size of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *m*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the last *m* rows of the *n*-by-*n* orthogonal matrix *Q*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine orgrq interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

tau — Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is ungrq.

?ungrq
Generates the complex matrix Q of the RQ factorization formed by ?gerqf*.*

Syntax

```
call cungrq(m, n, k, a, lda, tau, work, lwork, info)
call zungrq(m, n, k, a, lda, tau, work, lwork, info)
```

```
call ungrq(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates an *m*-by-*n* complex matrix with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors $H(i)$ of order *n*: $Q = H(1)^H * H(2)^H * \ldots * H(k)^H$ as returned by the routines gerqf. Use this routine after a call to cgerqf/zgerqf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q* ($m \geq 0$).
n	INTEGER. The number of columns of the matrix *Q* ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* ($m \geq k \geq 0$).
a, *tau*, *work*	REAL for cungrq DOUBLE PRECISION for zungrq Arrays: *a*(*lda*,*), *tau*(*), *work*(*lwork*). On entry, the (*m* - *k* + *i*)th row of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by cgerqf/zgerqf in the last *k* rows of its array argument *a*; *tau*(i) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by cgerqf/zgerqf; The second dimension of *a* must be at least max(1, *n*). The size of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *m*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *m* last rows of the *n*-by-*n* unitary matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `ungrq` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*m*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is orgrq.

?ormrq
Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by `?gerqf`*.*

Syntax

```
call sormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call ormrq(a, tau, c [,side] [,trans] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine multiplies a real *m*-by-*n* matrix *C* by *Q* or Q^T, where *Q* is the real orthogonal matrix defined as a product of *k* elementary reflectors H_i : $Q = H_1 H_2 \ldots H_k$ as returned by the *RQ* factorization routine gerqf.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (overwriting the result over *C*).

Input Parameters

side
CHARACTER*1. Must be either 'L' or 'R'.
If *side* = 'L', *Q* or Q^T is applied to *C* from the left.
If *side* = 'R', *Q* or Q^T is applied to *C* from the right.

trans
CHARACTER*1. Must be either 'N' or 'T'.
If *trans* = 'N', the routine multiplies *C* by *Q*.
If *trans* = 'T', the routine multiplies *C* by Q^T.

m
INTEGER. The number of rows in the matrix *C* (*m* ≥ 0).

n
INTEGER. The number of columns in *C* (*n* ≥ 0).

k
INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:
0 ≤ *k* ≤ *m*, if *side* = 'L';
0 ≤ *k* ≤ *n*, if *side* = 'R'.

a, *tau*, *c*, *work*
REAL for sormrq
DOUBLE PRECISION for dormrq.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*).
On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by sgerqf/dgerqf in the last *k* rows of its array argument *a*.
The second dimension of *a* must be at least max(1, *m*) if *side* = 'L', and at least max(1, *n*) if *side* = 'R'.
tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by sgerqf/dgerqf.
The size of *tau* must be at least max(1, *k*).
c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*)
work is a workspace array, its dimension max(1, *lwork*).

lda
INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *k*).

ldc
INTEGER. The leading dimension of *c*; *ldc* ≥ max(1, *m*).

lwork
INTEGER. The size of the *work* array. Constraints:
lwork ≥ max(1, *n*) if *side* = 'L';
lwork ≥ max(1, *m*) if *side* = 'R'.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ormrq interface are the following:

a	Holds the matrix *A* of size (*k*,*m*).
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is unmrq.

?unmrq
Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by ?gerqf.

Syntax

```
call cunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

```
call zunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call unmrq(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a complex *m*-by-*n* matrix *C* by *Q* or Q^H, where *Q* is the complex unitary matrix defined as a product of *k* elementary reflectors *H(i)* of order *n*: *Q = H(1)^H* H(2)^H*...*H(k)^H*as returned by the *RQ* factorization routine gerqf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products *Q*C*, Q^H**C*, *C*Q*, or *C**Q^H (overwriting the result over *C*).

Input Parameters

side — CHARACTER*1. Must be either 'L' or 'R'.

If *side* = 'L', *Q* or Q^H is applied to *C* from the left.

If *side* = 'R', *Q* or Q^H is applied to *C* from the right.

trans — CHARACTER*1. Must be either 'N' or 'C'.

If *trans* = 'N', the routine multiplies *C* by *Q*.

If *trans* = 'C', the routine multiplies *C* by Q^H.

m — INTEGER. The number of rows in the matrix *C* (*m*≥ 0).

n — INTEGER. The number of columns in *C* (*n*≥ 0).

k — INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

0 ≤*k*≤*m*, if *side* = 'L';

0 ≤*k*≤*n*, if *side* = 'R'.

a, *tau*, *c*, *work* — COMPLEX for cunmrq

DOUBLE COMPLEX for zunmrq.

Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).

On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by cgerqf/zgerqf in the last *k* rows of its array argument *a*.

The second dimension of *a* must be at least max(1, *m*) if *side* = 'L', and at least max(1, *n*) if *side* = 'R'.

tau(i) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by cgerqf/zgerqf.

The size of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.

The second dimension of *c* must be at least max(1, *n*)

work is a workspace array, its dimension `max(1, lwork)`.

`lda` `INTEGER`. The leading dimension of *a*; *lda*≥ max(1, *k*) .

`ldc` `INTEGER`. The leading dimension of *c*; *ldc*≥ max(1, *m*).

`lwork` `INTEGER`. The size of the *work* array. Constraints:

lwork≥ `max(1, n)` if `side = 'L'`;

lwork≥ `max(1, m)` if `side = 'R'`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

`c` Overwritten by the product $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).

`work(1)` If `info = 0`, on exit `work(1)` contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

`info` `INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `unmrq` interface are the following:

`a` Holds the matrix *A* of size (*k*,*m*).

`tau` Holds the vector of length (*k*).

`c` Holds the matrix *C* of size (*m*,*n*).

`side` Must be `'L'` or `'R'`. The default value is `'L'`.

`trans` Must be `'N'` or `'C'`. The default value is `'N'`.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ormrq.

?tzrzf
Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```
call stzrzf(m, n, a, lda, tau, work, lwork, info)
call dtzrzf(m, n, a, lda, tau, work, lwork, info)
call ctzrzf(m, n, a, lda, tau, work, lwork, info)
call ztzrzf(m, n, a, lda, tau, work, lwork, info)
call tzrzf(a [, tau] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces the *m*-by-*n* ($m \le n$) real/complex upper trapezoidal matrix *A* to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix $A = [A1\ A2] = [A_{1:m,\ 1:m},\ A_{1:m,\ m+1:n}]$ is factored as

$A = [R0]*Z$,

where *Z* is an *n*-by-*n* orthogonal/unitary matrix, *R* is an *m*-by-*m* upper triangular matrix, and *0* is the *m*-by-(*n*-*m*) zero matrix.

See larz that applies an elementary reflector returned by `?tzrzf` to a general matrix.

The `?tzrzf` routine replaces the deprecated `?tzrqf` routine.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* ($m \ge 0$).
n	INTEGER. The number of columns in *A* ($n \ge m$).
a, *work*	REAL for stzrzf DOUBLE PRECISION for dtzrzf COMPLEX for ctzrzf DOUBLE COMPLEX for ztzrzf. Arrays: *a*(*lda*,*) , *work*(*lwork*) .

The leading *m*-by-*n* upper trapezoidal part of the array *a* contains the matrix *A* to be factorized.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of *a*; at least max(1, *m*).

lwork — `INTEGER`. The size of the *work* array;

lwork≥ `max(1, m)`.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten on exit by the factorization data as follows:

the leading *m*-by-*m* upper triangular part of *a* contains the upper triangular matrix *R*, and elements *m* +1 to *n* of the first *m* rows of *a*, with the array *tau*, represent the orthogonal matrix *Z* as a product of *m* elementary reflectors.

tau — `REAL` for `stzrzf`

`DOUBLE PRECISION` for `dtzrzf`

`COMPLEX` for `ctzrzf`

`DOUBLE COMPLEX` for `ztzrzf`.

Array, size at least max (1, *m*). Contains scalar factors of the elementary reflectors for the matrix *Z*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.

If *info* = 0, the execution is successful.

If *info* = −*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `tzrzf` interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

tau — Holds the vector of length (*m*).

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is used to introduce zeros into the (*m* - *k* + 1)-th row of *A*, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where for real flavors

$$T(k) = I - tau * u(k) * u(k)^T, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

and for complex flavors

$$T(k) = I - tau * u(k) * u(k)^H, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and *z*(*k*) is an *l*-element vector. *tau* and *z*(*k*) are chosen to annihilate the elements of the *k*-th row of *A*2.

The scalar *tau* is returned in the *k*-th element of *tau* and the vector *u*(*k*) in the *k*-th row of *A*, such that the elements of *z*(*k*) are stored in `a(k, m+1), ..., a(k, n)`.

The elements of *R* are returned in the upper triangular part of *A*.

The matrix *Z* is given by

`Z = Z(1)*Z(2)*...*Z(m).`

For better performance, try using `lwork =m*blocksize`, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

ormrz	to apply matrix Q (for real matrices)
unmrz	to apply matrix Q (for complex matrices).

?ormrz
Multiplies a real matrix by the orthogonal matrix defined from the factorization formed by `?tzrzf`.

Syntax

```
call sormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call dormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call ormrz(a, tau, c, l [, side] [,trans] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The `?ormrz` routine multiplies a real *m*-by-*n* matrix *C* by *Q* or Q^T, where *Q* is the real orthogonal matrix defined as a product of *k* elementary reflectors `H(i)` of order *n*: `Q = H(1)* H(2)*...*H(k)` as returned by the factorization routine tzrzf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products `Q*C`, Q^T`*C`, `C*Q`, or `C*`Q^T (overwriting the result over *C*).

The matrix *Q* is of order *m* if `side = 'L'` and of order *n* if `side = 'R'`.

The `?ormrz` routine replaces the deprecated `?latzm` routine.

Input Parameters

side	`CHARACTER*1`. Must be either `'L'` or `'R'`. If `side = 'L'`, *Q* or Q^T is applied to *C* from the left. If `side = 'R'`, *Q* or Q^T is applied to *C* from the right.
trans	`CHARACTER*1`. Must be either `'N'` or `'T'`.

If *trans* = 'N', the routine multiplies *C* by *Q*.

If *trans* = 'T', the routine multiplies *C* by Q^T.

m — INTEGER. The number of rows in the matrix *C* (*m* ≥ 0).

n — INTEGER. The number of columns in *C* (*n* ≥ 0).

k — INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

0 ≤ *k* ≤ *m*, if *side* = 'L';

0 ≤ *k* ≤ *n*, if *side* = 'R'.

l — INTEGER.

The number of columns of the matrix *A* containing the meaningful part of the Householder reflectors. Constraints:

0 ≤ *l* ≤ *m*, if *side* = 'L';

0 ≤ *l* ≤ *n*, if *side* = 'R'.

a, *tau*, *c*, *work* — REAL for sormrz

DOUBLE PRECISION for dormrz.

Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*).

On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by stzrzf/dtzrzf in the last *k* rows of its array argument *a*.

The second dimension of *a* must be at least max(1, *m*) if *side* = 'L', and at least max(1, *n*) if *side* = 'R'.

tau(i) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by stzrzf/dtzrzf.

The size of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.

The second dimension of *c* must be at least max(1, *n*)

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *k*) .

ldc — INTEGER. The leading dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ormrz interface are the following:

a	Holds the matrix *A* of size (*k*,*m*).
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is unmrz.

?unmrz
Multiplies a complex matrix by the unitary matrix defined from the factorization formed by ?tzrzf.

Syntax

```
call cunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
```

```
call zunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call unmrz(a, tau, c, l [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a complex *m*-by-*n* matrix *C* by *Q* or Q^H, where *Q* is the unitary matrix defined as a product of *k* elementary reflectors `H(i)`:

`Q = H(1)^H* H(2)^H*...*H(k)^H` as returned by the factorization routine tzrzf.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products *Q***C*, Q^H**C*, *C***Q*, or *C**Q^H (overwriting the result over *C*).

The matrix *Q* is of order *m* if `side = 'L'` and of order *n* if `side = 'R'`.

Input Parameters

side
: CHARACTER*1. Must be either 'L' or 'R'.
If `side = 'L'`, *Q* or Q^H is applied to *C* from the left.
If `side = 'R'`, *Q* or Q^H is applied to *C* from the right.

trans
: CHARACTER*1. Must be either 'N' or 'C'.
If `trans = 'N'`, the routine multiplies *C* by *Q*.
If `trans = 'C'`, the routine multiplies *C* by Q^H.

m
: INTEGER. The number of rows in the matrix *C* (*m*≥ 0).

n
: INTEGER. The number of columns in *C* (*n*≥ 0).

k
: INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:
0 ≤*k*≤*m*, if `side = 'L'`;
0 ≤*k*≤*n*, if `side = 'R'`.

l
: INTEGER.
The number of columns of the matrix *A* containing the meaningful part of the Householder reflectors. Constraints:
0 ≤*l*≤*m*, if `side = 'L'`;
0 ≤*l*≤*n*, if `side = 'R'`.

a, *tau*, *c*, *work*
: COMPLEX for cunmrz
DOUBLE COMPLEX for zunmrz.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).
On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector `H(i)`, for i = 1,2,...,*k*, as returned by ctzrzf/ztzrzf in the last *k* rows of its array argument *a*.

The second dimension of *a* must be at least max(1, *m*) if `side = 'L'`, and at least max(1, *n*) if `side = 'R'`.

tau(i) must contain the scalar factor of the elementary reflector `H(i)`, as returned by `ctzrzf`/`ztzrzf`.

The size of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.

The second dimension of *c* must be at least max(1, *n*)

work is a workspace array, its dimension `max(1, lwork)`.

`lda` INTEGER. The leading dimension of *a*; `lda` ≥ `max(1, k)`.

`ldc` INTEGER. The leading dimension of *c*; `ldc` ≥ `max(1, m)`.

`lwork` INTEGER. The size of the *work* array. Constraints:

`lwork` ≥ `max(1, n)` if `side = 'L'`;

`lwork` ≥ `max(1, m)` if `side = 'R'`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

`c` Overwritten by the product $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).

`work(1)` If `info = 0`, on exit `work(1)` contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

`info` INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `unmrz` interface are the following:

`a` Holds the matrix *A* of size (*k*,*m*).

`tau` Holds the vector of length (*k*).

`c` Holds the matrix *C* of size (*m*,*n*).

`side` Must be `'L'` or `'R'`. The default value is `'L'`.

`trans` Must be `'N'` or `'C'`. The default value is `'N'`.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ormrz.

?ggqrf

Computes the generalized QR factorization of two matrices.

Syntax

```
call sggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call ggqrf(a, b [,taua] [,taub] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine forms the generalized *QR* factorization of an *n*-by-*m* matrix *A* and an *n*-by-*p* matrix *B* as `A = Q*R, B = Q*T*Z`, where *Q* is an *n*-by-*n* orthogonal/unitary matrix, *Z* is a *p*-by-*p* orthogonal/unitary matrix, and *R* and *T* assume one of the forms:

$$R = \begin{array}{c} \\ m \\ n-m \end{array} \begin{array}{c} m \\ \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \end{array}, \quad \text{if } n \geq m$$

or

$$R = \begin{array}{c} \\ n \end{array} \begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} \end{array}, \quad \text{if } n < m$$

where R_{11} is upper triangular, and

$$T = \begin{array}{c} \\ n \end{array} \begin{array}{c} \begin{array}{cc} p-n & n \end{array} \\ \begin{pmatrix} 0 & T_{12} \end{pmatrix} \end{array}, \quad \text{if } n \leq p,$$

$$T = \begin{array}{c} \\ n \\ p-n \end{array} \begin{array}{c} \begin{array}{c} p \end{array} \\ \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \end{array}, \quad \text{if } n > p,$$

where T_{12} or T_{21} is a p-by-p upper triangular matrix.

In particular, if B is square and nonsingular, the GQR factorization of A and B implicitly gives the QR factorization of $B^{-1}A$ as:

$B^{-1}*A = Z^{T}*(T^{-1}*R)$ (for real flavors) or $B^{-1}*A = Z^{H}*(T^{-1}*R)$ (for complex flavors).

Input Parameters

n	INTEGER. The number of rows of the matrices *A* and *B* (*n*≥ 0).
m	INTEGER. The number of columns in *A* (*m*≥ 0).
p	INTEGER. The number of columns in *B* (*p*≥ 0).
a, *b*, *work*	REAL for sggqrf DOUBLE PRECISION for dggqrf COMPLEX for cggqrf DOUBLE COMPLEX for zggqrf. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *m*). *b*(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *p*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The leading dimension of *b*; at least max(1, *n*).

lwork — INTEGER. The size of the *work* array; must be at least max(1, *n*, *m*, *p*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a, *b* — Overwritten by the factorization data as follows:

on exit, the elements on and above the diagonal of the array *a* contain the min(*n*,*m*)-by-*m* upper trapezoidal matrix *R* (*R* is upper triangular if $n \geq m$);the elements below the diagonal, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors ;

if $n \leq p$, the upper triangle of the subarray *b*(1:*n*, *p*-*n*+1:*p*) contains the *n*-by-*n* upper triangular matrix *T*;

if *n* > *p*, the elements on and above the (*n*-*p*)th subdiagonal contain the *n*-by-*p* upper trapezoidal matrix *T*; the remaining elements, with the array *taub*, represent the orthogonal/unitary matrix *Z* as a product of elementary reflectors.

taua, *taub* — REAL for sggqrf

DOUBLE PRECISION for dggqrf

COMPLEX for cggqrf

DOUBLE COMPLEX for zggqrf.

Arrays, size at least max (1, min(*n*, *m*)) for *taua* and at least max (1, min(*n*, *p*)) for *taub*. The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*.

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ggqrf interface are the following:

a — Holds the matrix *A* of size (*n*,*m*).

b — Holds the matrix *B* of size (*n*,*p*).

taua	Holds the vector of length min(*n,m*).
taub	Holds the vector of length min(*n,p*).

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1)*H*(2)...*H*(*k*), where *k* = min(n,m).

Each *H*(i) has the form

$H(i) = I - \tau_a * v * v^T$ for real flavors, or

$H(i) = I - \tau_a * v * v^H$ for complex flavors,

where τ_a is a real/complex scalar, and *v* is a real/complex vector with $v_j = 0$ for $1 \le j \le i - 1$, $v_i = 1$.

On exit, for$i + 1 \le j \le n$, v_j is stored in *a*(i+1:*n*, i) and τ_a is stored in *taua*(*i*)

The matrix *Z* is represented as a product of elementary reflectors

Z = *H*(1)*H*(2)...*H*(*k*), where *k* = min(n,p).

Each *H*(i) has the form

$H(i) = I - \tau_b * v * v^T$ for real flavors, or

$H(i) = I - \tau_b * v * v^H$ for complex flavors,

where τ_b is a real/complex scalar, and *v* is a real/complex vector with $v_{p-k+1} = 1$, $v_j = 0$ for $p - k + 1 \le j \le p - 1$, .

On exit, for $1 \le j \le p - k + i - 1$, v_j is stored in *b*(n-k+i, 1:p-k+i-1) and τ_b is stored in *taub*(*i*).

For better performance, try using *lwork*≥ max(*n*,*m*, *p*)*max(*nb1*,*nb2*,*nb3*), where *nb1* is the optimal blocksize for the *QR* factorization of an *n*-by-*m* matrix, *nb2* is the optimal blocksize for the *RQ* factorization of an *n*-by-*p* matrix, and *nb3* is the optimal blocksize for a call of ormqr/unmqr.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggrqf
Computes the generalized RQ factorization of two matrices.

Syntax

```
call sggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
```

```
call ggrqf(a, b [,taua] [,taub] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the generalized *RQ* factorization of an *m*-by-*n* matrix *A* and an *p*-by-*n* matrix *B* as `A = R*Q`, `B = Z*T*Q`, where *Q* is an *n*-by-*n* orthogonal/unitary matrix, *Z* is a *p*-by-*p* orthogonal/unitary matrix, and *R* and *T* assume one of the forms:

$$R = \begin{array}{c} \\ m \end{array} \begin{array}{c} \begin{array}{cc} n-m & m \end{array} \\ \begin{pmatrix} 0 & R_{12} \end{pmatrix} \end{array}, \quad \text{if } m \le n,$$

or

$$R = \begin{array}{c} \\ m-n \\ n \end{array} \begin{array}{c} n \\ \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix} \end{array}, \quad \text{if } m > n,$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{array}{c} \\ n \\ p-n \end{array} \begin{array}{c} n \\ \begin{pmatrix} T_{11} \\ 0 \end{pmatrix} \end{array}, \quad \text{if } p \ge n,$$

or

$$T = \begin{array}{c} \\ p \end{array} \begin{array}{c} \begin{array}{cc} p & n-p \end{array} \\ \begin{pmatrix} T_{11} & T_{12} \end{pmatrix} \end{array}, \quad \text{if } p < n,$$

where T_{11} is upper triangular.

In particular, if *B* is square and nonsingular, the *GRQ* factorization of *A* and *B* implicitly gives the *RQ* factorization of $A*B^{-1}$ as:

$A*B^{-1} = (R*T^{-1})*Z^{T}$ (for real flavors) or $A*B^{-1} = (R*T^{-1})*Z^{H}$ (for complex flavors).

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m*≥ 0).
p	INTEGER. The number of rows in *B* (*p*≥ 0).
n	INTEGER. The number of columns of the matrices *A* and *B* (*n*≥ 0).
a, *b*, *work*	REAL for sggrqf DOUBLE PRECISION for dggrqf COMPLEX for cggrqf DOUBLE COMPLEX for zggrqf. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *p*-by-*n* matrix *B*. The second dimension of *b* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The leading dimension of *b*; at least max(1, *p*).
lwork	INTEGER. The size of the *work* array; must be at least max(1, *n*, *m*, *p*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a, *b*	Overwritten by the factorization data as follows: on exit, if *m*≤*n*, the upper triangle of the subarray *a*(1:*m*, *n*-*m*+1:*n*) contains the *m*-by-*m* upper triangular matrix *R*; if *m* > *n*, the elements on and above the (*m*-*n*)th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*; the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. The elements on and above the diagonal of the array *b* contain the min(*p*,*n*)-by-*n* upper trapezoidal matrix *T* (*T* is upper triangular if *p*≥*n*); the elements below the diagonal, with the array *taub*, represent the orthogonal/unitary matrix *Z* as a product of elementary reflectors.
taua, *taub*	REAL for sggrqf DOUBLE PRECISION for dggrqf COMPLEX for cggrqf

DOUBLE COMPLEX for zggrqf.

Arrays, size at least max (1, min(*m*, *n*)) for *taua* and at least max (1, min(*p*, *n*)) for *taub*.

The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*.

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ggrqf interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

b — Holds the matrix *A* of size (*p*,*n*).

taua — Holds the vector of length min(*m*,*n*).

taub — Holds the vector of length min(*p*,*n*).

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1)*H*(2)...*H*(*k*), where *k* = min(m,n).

Each *H*(i) has the form

H(i) = *I* - *taua***v***v*^{T} for real flavors, or

H(i) = *I* - *taua***v***v*^{H} for complex flavors,

where *taua* is a real/complex scalar, and *v* is a real/complex vector with $v_{n-k+i} = 1$, $v_{n-k+i+1:n} = 0$.

On exit, $v_{1:n-k+i-1}$ is stored in *a*(m-k+i,1:n-k+i-1) and *taua* is stored in *taua*(*i*).

The matrix *Z* is represented as a product of elementary reflectors

Z = *H*(1)*H*(2)...*H*(*k*), where *k* = min(p,n).

Each *H*(i) has the form

H(i) = *I* - *taub***v***v*^{T} for real flavors, or

H(i) = *I* - *taub***v***v*^{H} for complex flavors,

where *taub* is a real/complex scalar, and *v* is a real/complex vector with $v_{1:i-1} = 0$, $v_i = 1$.

On exit, $v_{i+1:p}$ is stored in *b*(i+1:p, i) and *taub* is stored in *taub*(*i*).

For better performance, try using

lwork≥ max(*n*,*m*, *p*)*max(*nb1*,*nb2*,*nb3*),

where *nb1* is the optimal blocksize for the *RQ* factorization of an *m*-by-*n* matrix, *nb2* is the optimal blocksize for the *QR* factorization of an *p*-by-*n* matrix, and *nb3* is the optimal blocksize for a call of ?ormrq/?unmrq.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork*= -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork*= -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tpqrt
Computes a blocked QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q.

Syntax

```
call stpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call dtpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call ctpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call ztpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call tpqrt(a, b, t, nb[, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The input matrix *C* is an (*n*+*m*)-by-*n* matrix

$$C = \begin{bmatrix} A \\ B \end{bmatrix} \begin{matrix} \leftarrow n \times n \text{ upper triangular} \\ \leftarrow m \times n \text{ pentagonal} \end{matrix}$$

where *A* is an *n*-by-*n* upper triangular matrix, and *B* is an *m*-by-*n* pentagonal matrix consisting of an (*m*-*l*)-by-*n* rectangular matrix *B1* on top of an *l*-by-*n* upper trapezoidal matrix *B2*:

$$B = \begin{bmatrix} B1 \\ B2 \end{bmatrix} \begin{matrix} \leftarrow (m-l)\times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{matrix}$$

The upper trapezoidal matrix *B2* consists of the first `l` rows of an `n`-by-`n` upper triangular matrix, where 0 ≤ `l` ≤ min(`m`,`n`). If `l`=0, *B* is an `m`-by-`n` rectangular matrix. If `m`=`l`=`n`, *B* is upper triangular. The elementary reflectors *H(i)* are stored in the *i*th column below the diagonal in the (`n`+`m`)-by-`n` input matrix *C*. The structure of vectors defining the elementary reflectors is illustrated by:

$$\begin{bmatrix} I \\ V \end{bmatrix} \begin{matrix} \leftarrow n \times n \text{ identity} \\ \leftarrow m \times n \text{ pentagonal} \end{matrix}$$

The elements of the unit matrix *I* are not stored. Thus, *V* contains all of the necessary information, and is returned in array `b`.

NOTE
Note that *V* has the same form as *B*:

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix} \begin{matrix} \leftarrow (m-l)\times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{matrix}$$

The columns of *V* represent the vectors which define the *H(i)*s.

The number of blocks is *k* = ceiling(`n`/`nb`), where each block is of order `nb` except for the last block, which is of order *ib* = `n` - (*k*-1)*`nb`. For each of the *k* blocks, an upper triangular block reflector factor is computed: *T1*, *T2*, ..., *Tk*. The `nb`-by-`nb` (*ib*-by-*ib* for the last block) *Ti*s are stored in the `nb`-by-`n` array `t` as

`t = [T1T2 ... Tk]` .

Input Parameters

`m` — `INTEGER`. The total number of rows in the matrix *B* (`m` ≥ 0).

`n` — `INTEGER`. The number of columns in *B* and the order of the triangular matrix *A* (`n` ≥ 0).

`l` — `INTEGER`. The number of rows of the upper trapezoidal part of *B* (min(`m`, `n`) ≥ `l` ≥ 0).

nb	INTEGER. The block size to use in the blocked *QR* factorization ($n \geq nb \geq 1$).
a, *b*, *work*	REAL for stpqrt DOUBLE PRECISION for dtpqrt COMPLEX for ctpqrt COMPLEX*16 for ztpqrt. Arrays: *a* size (*lda*, *n*) contains the *n*-by-*n* upper triangular matrix *A*. *b* size (*ldb*, *n*), the pentagonal *m*-by-*n* matrix *B*. The first (*m*-*l*) rows contain the rectangular *B1* matrix, and the next *l* rows contain the upper trapezoidal *B2* matrix. *work* size (*nb*, *n*) is a workspace array.
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; at least max(1, *m*).
ldt	INTEGER. The leading dimension of *t*; at least *nb*.

Output Parameters

a	The elements on and above the diagonal of the array contain the upper triangular matrix *R*.
b	The pentagonal matrix *V*.
t	REAL for stpqrt DOUBLE PRECISION for dtpqrt COMPLEX for ctpqrt COMPLEX*16 for ztpqrt. Array, size (*ldt*, *n*). The upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks.
info	INTEGER. If *info* = 0, the execution is successful. If *info* < 0 and *info* = -*i*, the *i*th argument had an illegal value.

?tpmqrt

Applies a real or complex orthogonal matrix obtained from a "triangular-pentagonal" complex block reflector to a general real or complex matrix, which consists of two blocks.

Syntax

```
call stpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call dtpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call ctpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
```

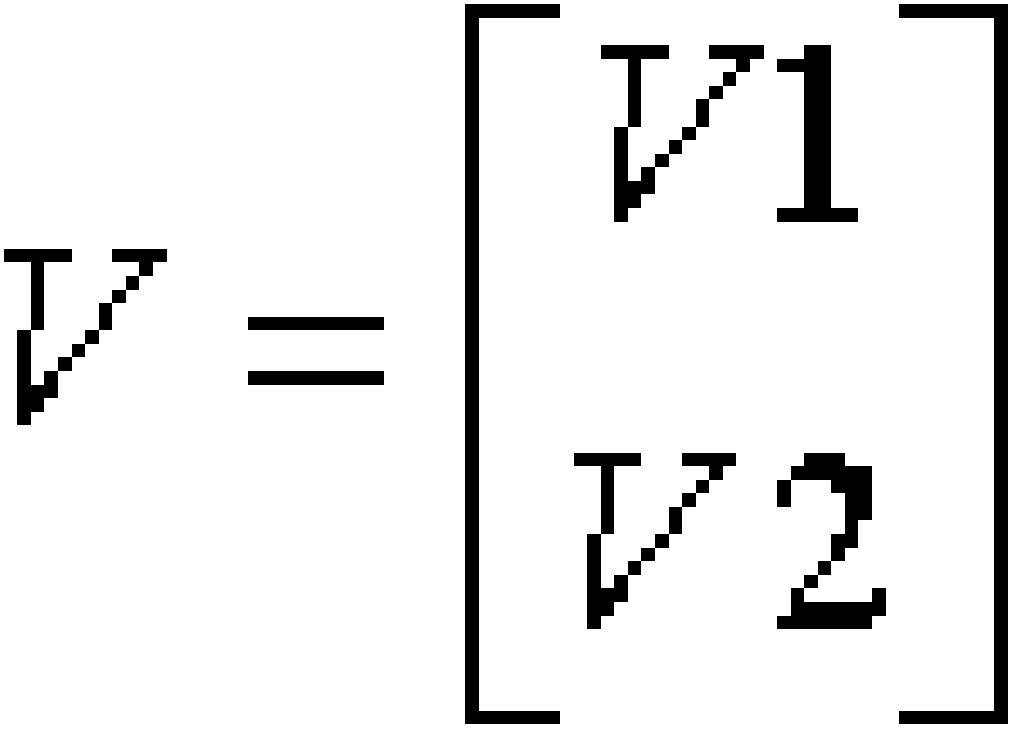
```
call ztpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call tpmqrt( v, t, a, b, k, nb[, trans][, side][, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The columns of the pentagonal matrix *V* contain the elementary reflectors *H*(1), *H*(2), ..., *H*(*k*); *V* is composed of a rectangular block *V1* and a trapezoidal block *V2*:

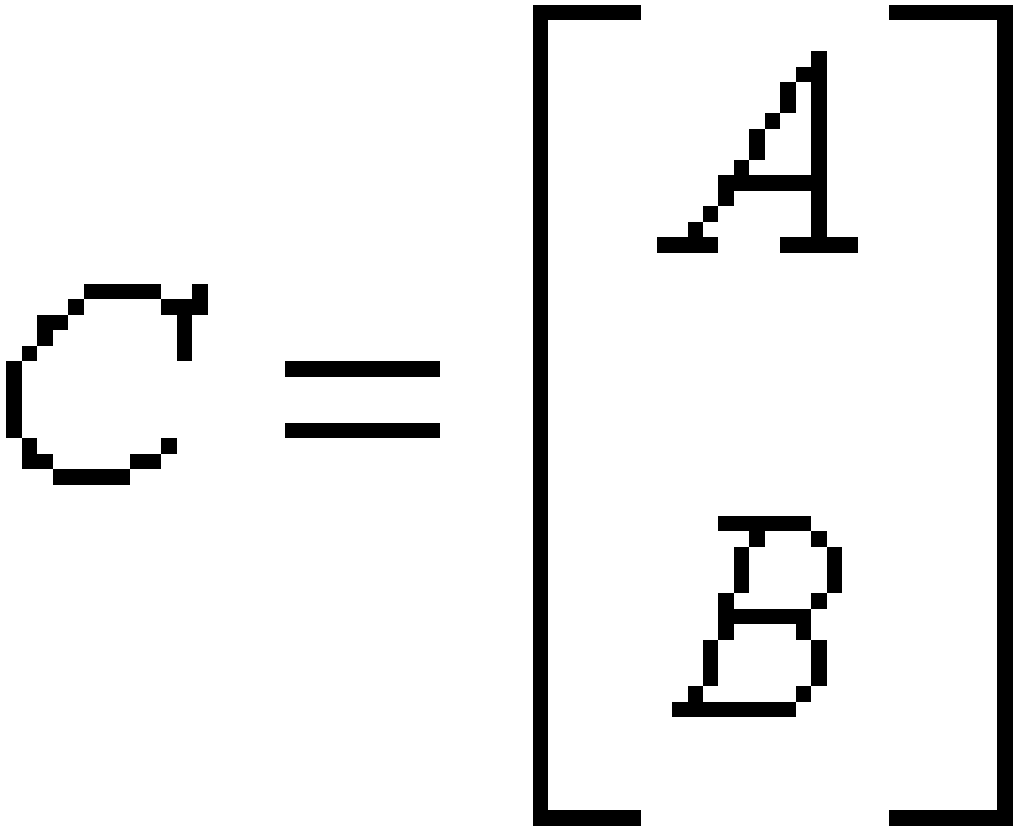


The size of the trapezoidal block *V2* is determined by the parameter *l*, where 0 ≤ *l* ≤ *k*. *V2* is upper trapezoidal, consisting of the first *l* rows of a *k*-by-*k* upper triangular matrix.

If *l*=*k*, *V2* is upper triangular;

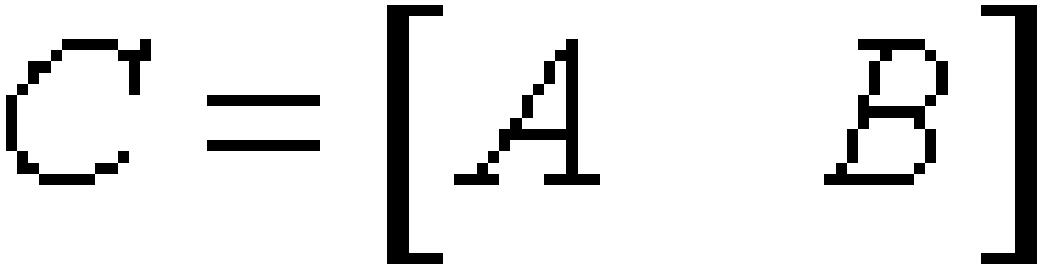
If *l*=0, there is no trapezoidal block, so *V* = *V1* is rectangular.

If *side* = 'L':



where *A* is *k*-by-*n*, *B* is *m*-by-*n* and *V* is *m*-by-*k*.

If *side* = 'R':



where *A* is *m*-by-*k*, *B* is *m*-by-*n* and *V* is *n*-by-*k*.

The real/complex orthogonal matrix *Q* is formed from *V* and *T*.

If *trans*='N' and *side*='L', *c* contains *Q* * *C* on exit.

If *trans*='T' and *side*='L', *C* contains Q^T * *C* on exit.

If *trans*='C' and *side*='L', *C* contains Q^H * *C* on exit.

If *trans*='N' and *side*='R', *C* contains *C* * *Q* on exit.

If *trans*='T' and *side*='R', *C* contains *C* * Q^T on exit.

If *trans*='C' and *side*='R', *C* contains $C * Q^H$ on exit.

Input Parameters

side CHARACTER*1

='L': apply Q, Q^T, or Q^H from the left.

='R': apply Q, Q^T, or Q^H from the right.

trans CHARACTER*1

='N', no transpose, apply Q.

='T', transpose, apply Q^T.

='C', transpose, apply Q^H.

m INTEGER. The number of rows in the matrix B, $(m \geq 0)$.

n INTEGER. The number of columns in the matrix B, $(n \geq 0)$.

k INTEGER. The number of elementary reflectors whose product defines the matrix Q, $(k \geq 0)$.

l INTEGER. The order of the trapezoidal part of V ($k \geq l \geq 0$).

nb INTEGER.

The block size used for the storage of *t*, $k \geq nb \geq 1$. This must be the same value of *nb* used to generate *t* in tpqrt.

v REAL for stpmqrt

DOUBLE PRECISION for dtpmqrt

COMPLEX for ctpmqrt

COMPLEX*16 for ztpmqrt.

Size (*ldv*, *k*)

The *i*th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1,2,...,k$, as returned by tpqrt in array argument *b*.

ldv INTEGER. The leading dimension of the array *v*.

If *side* = 'L', *ldv* must be at least max(1,*m*);

If *side* = 'R', *ldv* must be at least max(1,*n*).

t REAL for stpmqrt

DOUBLE PRECISION for dtpmqrt

COMPLEX for ctpmqrt

COMPLEX*16 for ztpmqrt.

Array, size (*ldt*, *k*).

The upper triangular factors of the block reflectors as returned by tpqrt, stored as an *nb*-by-*k* matrix.

ldt INTEGER. The leading dimension of the array *t*. *ldt* must be at least *nb*.

a
REAL for stpmqrt
DOUBLE PRECISION for dtpmqrt
COMPLEX for ctpmqrt
COMPLEX*16 for ztpmqrt.
If *side* = 'L', size (*lda*, *n*).
If *side* = 'R', size (*lda*, *k*).
The *k*-by-*n* or *m*-by-*k* matrix *A*.

lda
INTEGER. The leading dimension of the array *a*.
If *side* = 'L', *lda* must be at least max(1,*k*).
If *side* = 'R', *lda* must be at least max(1,*m*).

b
REAL for stpmqrt
DOUBLE PRECISION for dtpmqrt
COMPLEX for ctpmqrt
COMPLEX*16 for ztpmqrt.
Size (*ldb*, *n*).
The *m*-by-*n* matrix *B*.

ldb
INTEGER. The leading dimension of the array *b*. *ldb* must be at least max(1,*m*).

work
REAL for stpmqrt
DOUBLE PRECISION for dtpmqrt
COMPLEX for ctpmqrt
COMPLEX*16 for ztpmqrt.
Workspace array. If *side* = 'L' DIMENSION *n***nb*. If *side* = 'R' DIMENSION *m***nb*.

Output Parameters

a
Overwritten by the corresponding block of the product $Q*C$, $C*Q$, Q^T*C, $C*Q^T$, Q^H*C, or $C*Q^H$.

b
Overwritten by the corresponding block of the product $Q*C$, $C*Q$, Q^T*C, $C*Q^T$, Q^H*C, or $C*Q^H$.

info
(global) INTEGER.
= 0: the execution is successful.
< 0: if *info* = -*i*, the *i*th argument had an illegal value.

?tplqt
Computes a blocked LQ factorization of a complex "triangular-pentagonal" matrix composed of a triangular block A and pentagonal block B, using the compact WY representation for Q.

```
call stplqt(m, n, l, mb, a, lda, b, ldb, t, ldt, work, info)
call dtplqt(m, n, l, mb, a, lda, b, ldb, t, ldt, work, info)
call ctplqt(m, n, l, mb, a, lda, b, ldb, t, ldt, work, info)
call ztplqt(m, n, l, mb, a, lda, b, ldb, t, ldt, work, info)
```

Description

?tplqt computes a blocked LQ factorization of a real or complex "triangular-pentagonal" matrix *C*, which is composed of a triangular block *A* and pentagonal block *B*, using the compact WY representation for *Q*.

The input matrix *C* is an *m*-by-(*m*+*n*) matrix:

C = [*A*] [*B*]

where *A* is a lower triangular *m*-by-*m* matrix, and *B* is an *m*-by-*n* pentagonal matrix consisting of an *m*-by-(*n*-*l*) rectangular matrix *B1* to the left of an *m*-by-*l* lower trapezoidal matrix *B2*:

[*B*] = [*B1*] [*B2*]

[*B1*] <- *m*-by-(*n*-*l*) rectangular

[*B2*] <- *m*-by-*l* lower trapezoidal.

The lower trapezoidal matrix *B2* consists of the first *l* columns of an *m*-by-*m* lower triangular matrix, where 0 ≤*l*≤ min(*m*,*n*). If *l*=0, *b* is rectangular *m*-by-*n*; if *m*=*l*=*n*, *b* is lower triangular.

The matrix *W* stores the elementary reflectors *H*(*i*) in the *i*-th row above the diagonal (of *A*) in the *m*-by-(*m* +*n*) input matrix *C*:

[*C*] = [*A*] [*B*]

[*A*] <- lower triangular *m*-by-*m*

[*B*] <- *m*-by-*n* pentagonal

so that *W* can be represented as

[*W*] = [*I*] [*V*]

[*I*] <- *m*-by-*m* identity matrix

[*V*] <- *m*-by-*n*, same form as *B*.

Thus, all of information needed for *W* is contained on exit in the array *b*, called *V* in the preceding. Note that *V* has the same form as *B*; that is,

[*V*] = [*V1*] [*V2*]

[*V1*] <- *m*-by-(*n*-*l*) rectangular

[*V2*] <- *m*-by-*l* lower trapezoidal.

The rows of *V* represent the vectors which define the *H*(*i*) elementary reflectors .

The number of blocks is *B* = ceiling(*m*/*mb*), where each block is of order *mb* except for the last block, which is of order *ib* = *m* - (*m* - 1)**mb*. For each of the *B* blocks, a upper triangular block reflector factor is computed:

T1, *T2*, ..., *TB*.

The *mb*-by-*mb* (and *ib*-by-*ib* for the last block) *Ti*s are stored in the *mb*-by-*n* array *t* as

T = [*T1T2* ... *TB*].

Input Parameters

m	INTEGER. The number of rows of the matrix *B*, and the order of the triangular matrix *A*. *m*≥ 0.
n	INTEGER. The number of columns of the matrix *B*. *n*≥ 0.
l	INTEGER. The number of rows of the lower trapezoidal part of *B*. min(*m*, *n*) ≥*l*≥ 0.
mb	INTEGER. The block size to be used in the blocked QR. *m*≥*mb*≥ 1.
a	REAL for stplqt DOUBLE PRECISION for dtplqt COMPLEX for ctplqt COMPLEX*16 for ztplqt Array of size (*lda*,*m*). On entry, the lower triangular *m*-by-*m* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*m*).
b	REAL for stplqt DOUBLE PRECISION for dtplqt COMPLEX for ctplqt COMPLEX*16 for ztplqt Array of size (*ldb*,*n*). On entry, the pentagonal *m*-by-*n* matrix *B*. The first *n*-*l* columns are rectangular, and the last *l* columns are lower trapezoidal.
ldb	INTEGER. The leading dimension of the array *b*. *ldb*≥ max(1,*m*).
ldt	INTEGER. The leading dimension of the array *t*. *ldt*≥*mb*.

Output Parameters

a	On exit, the elements on and below the diagonal of the array contain the lower triangular matrix *L*.
b	On exit, *b* contains the pentagonal matrix *V*.
t	REAL for stplqt DOUBLE PRECISION for dtplqt COMPLEX for ctplqt COMPLEX*16 for ztplqt Array of size (*ldt*,*n*). The lower triangular block reflectors stored in compact form as a sequence of upper triangular blocks.
work	REAL for stplqt DOUBLE PRECISION for dtplqt COMPLEX for ctplqt

	COMPLEX*16 for ztplqt Array of size (*mb***m*).
info	INTEGER. *info* = 0: successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?tpmlqt

Applies an orthogonal matrix obtained from a "triangular-pentagonal" block reflector to a general matrix.

```
call stpmlqt(side, trans, m, n, k, l, mb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call dtpmlqt(side, trans, m, n, k, l, mb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call ctpmlqt(side, trans, m, n, k, l, mb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call ztpmlqt(side, trans, m, n, k, l, mb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
```

Description

?tpmlqt applies an orthogonal matrix *Q* obtained from a "triangular-pentagonal" block reflector *H* to a general matrix *C*, which consists of two blocks *A* and *B*.

The columns of the pentagonal matrix *V* contain the elementary reflectors *H*(1), *H*(2), ..., *H*(*k*); *V* is composed of a rectangular block *V1* and a trapezoidal block *V2*:

V = [*V1*] [*V2*].

The size of the trapezoidal block *V2* is determined by the parameter *l*, where 0 ≤*l*≤*k*; *V2* is lower trapezoidal, consisting of the first *l* rows of a *k*-by-*k* upper triangular matrix. If *l*=*k*, *V2* is lower triangular; if *l*=0, there is no trapezoidal block, hence *V* = *V1* is rectangular.

If *side* = 'L':

C = [*A*] [*B*]

where *A* is *k*-by-*n*, *B* is *m*-by-*n*, and *C* is *k*-by-*m*.

If *side* = 'R':

C = [*AB*]

where *A* is *m*-by-*k*, *B* is *m*-by-*n*, and *C* is *k*-by-*n*.

The real orthogonal matrix *Q* is formed from *V* and *T*.

If *trans*='N' and *side*='L', *C* is on exit replaced with *Q* * *C*.

C is on exit replaced with

	side = 'L'	*side* = 'R'
trans = 'N'	*Q***C*	*C***Q*
trans = 'T'	Q^T**C*	*C**Q^T
trans = 'C'	Q^H**C*	*C**Q^H

Input Parameters

side CHARACTER*1.

= 'L': apply op(Q) from the left;

= 'R': apply op(Q) from the right.

trans CHARACTER*1.

= 'N': No transpose, op(Q) = Q;

= 'T': Transpose, op(Q) = Q^T;

= 'C': Transpose, op(Q) = Q^H.

m INTEGER. The number of rows of the matrix B. $m \geq 0$.

n INTEGER. The number of columns of the matrix B. $n \geq 0$.

k INTEGER. The number of elementary reflectors whose product defines the matrix Q.

l INTEGER. The order of the trapezoidal part of V. $k \geq l \geq 0$.

mb INTEGER. The block size used for the storage of T. $k \geq mb \geq 1$. This must be the same value of *mb* used to generate T in ?tplqt.

v REAL for stpmlqt

DOUBLE PRECISION for dtpmlqt

COMPLEX for ctpmlqt

COMPLEX*16 for ztpmlqt

Array of size (*lda*,*k*). The i-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1,2,...,k$, as returned by ?tplqt in *b*.

ldv INTEGER. The leading dimension of the array *v*. If *side* = 'L', $ldv \geq \max(1, m)$; if *side* = 'R', $ldv \geq \max(1, n)$.

t REAL for stpmlqt

DOUBLE PRECISION for dtpmlqt

COMPLEX for ctpmlqt

COMPLEX*16 for ztpmlqt

Array of size (*ldt*,*k*). The upper triangular factors of the block reflectors as returned by ?tplqt, stored as a *mb*-by-*k* matrix.

ldt INTEGER. The leading dimension of the array *t*. $ldt \geq mb$.

a REAL for stpmlqt

DOUBLE PRECISION for dtpmlqt

COMPLEX for ctpmlqt

COMPLEX*16 for ztpmlqt

Array of size (*lda*,*n*) if *side* = 'L', or size (*lda*,*k*) if *side* = 'R'.

On entry, the *k*-by-*n* or *m*-by-*k* matrix A.

lda	INTEGER. The leading dimension of the array *a*. If *side* = 'L', LDC ≥ max(1, *k*); if *side* = 'R', LDC ≥ max(1, *m*).
b	REAL for stpmlqt DOUBLE PRECISION for dtpmlqt COMPLEX for ctpmlqt COMPLEX*16 for ztpmlqt Array of size (*ldb*, *n*). On entry, the *m*-by-*n* matrix *B*.
ldb	INTEGER. The leading dimension of the array *b*. *ldb*≥ max(1, *m*).

Output Parameters

a	On exit, *a* is overwritten by the corresponding block of op(*Q*)**C* or *C**op(*Q*) See Description.
b	REAL for stpmlqt DOUBLE PRECISION for dtpmlqt COMPLEX for ctpmlqt COMPLEX*16 for ztpmlqt On exit, *b* is overwritten by the corresponding block of op(*Q*)**C* or *C**op(*Q*) See Description.
work	REAL for stpmlqt DOUBLE PRECISION for dtpmlqt COMPLEX for ctpmlqt COMPLEX*16 for ztpmlqt Array. The size of *work* is *n***mb* if *side* = 'L', or *m***mb* if *side* = 'R'.
info	INTEGER. *info* = 0: successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

Singular Value Decomposition: LAPACK Computational Routines

This topic describes LAPACK routines for computing the *singular value decomposition* (SVD) of a general *m*-by-*n* matrix *A*:

$A = U\Sigma V^H$.

In this decomposition, *U* and *V* are unitary (for complex *A*) or orthogonal (for real *A*); Σ is an *m*-by-*n* diagonal matrix with real diagonal elements σ_i:

$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m, n)} \geq 0$.

The diagonal elements σ_i are *singular values* of *A*. The first $\min(m, n)$ columns of the matrices *U* and *V* are, respectively, *left* and *right singular vectors* of *A*. The singular values and singular vectors satisfy

$Av_i = \sigma_i u_i$ and $A^H u_i = \sigma_i v_i$

where u_i and v_i are the *i*-th columns of *U* and *V*, respectively.

To find the SVD of a general matrix *A*, call the LAPACK routine `?gebrd` or `?gbbrd` for reducing *A* to a bidiagonal matrix *B* by a unitary (orthogonal) transformation: $A = QBP^H$. Then call `?bdsqr`, which forms the SVD of a bidiagonal matrix: $B = U_1 \Sigma V_1^H$.

Thus, the sought-for SVD of *A* is given by $A = U \Sigma V^H = (QU_1) \Sigma (V_1^H P^H)$.

Table "Computational Routines for Singular Value Decomposition (SVD)" lists LAPACK routines (FORTRAN 77 interface) that perform singular value decomposition of matrices. The corresponding routine names in the Fortran 95 interface are the same except that the first character is removed.

Computational Routines for Singular Value Decomposition (SVD)

Operation	Real matrices	Complex matrices
Reduce *A* to a bidiagonal matrix *B*: $A = QBP^H$ (full storage)	?gebrd	?gebrd
Reduce *A* to a bidiagonal matrix *B*: $A = QBP^H$ (band storage)	?gbbrd	?gbbrd
Generate the orthogonal (unitary) matrix *Q* or P	?orgbr	?ungbr
Apply the orthogonal (unitary) matrix *Q* or P	?ormbr	?unmbr
Form singular value decomposition of the bidiagonal matrix *B*: $B = U \Sigma V^H$	?bdsqr ?bdsdc	?bdsqr

Decision Tree: Singular Value Decomposition

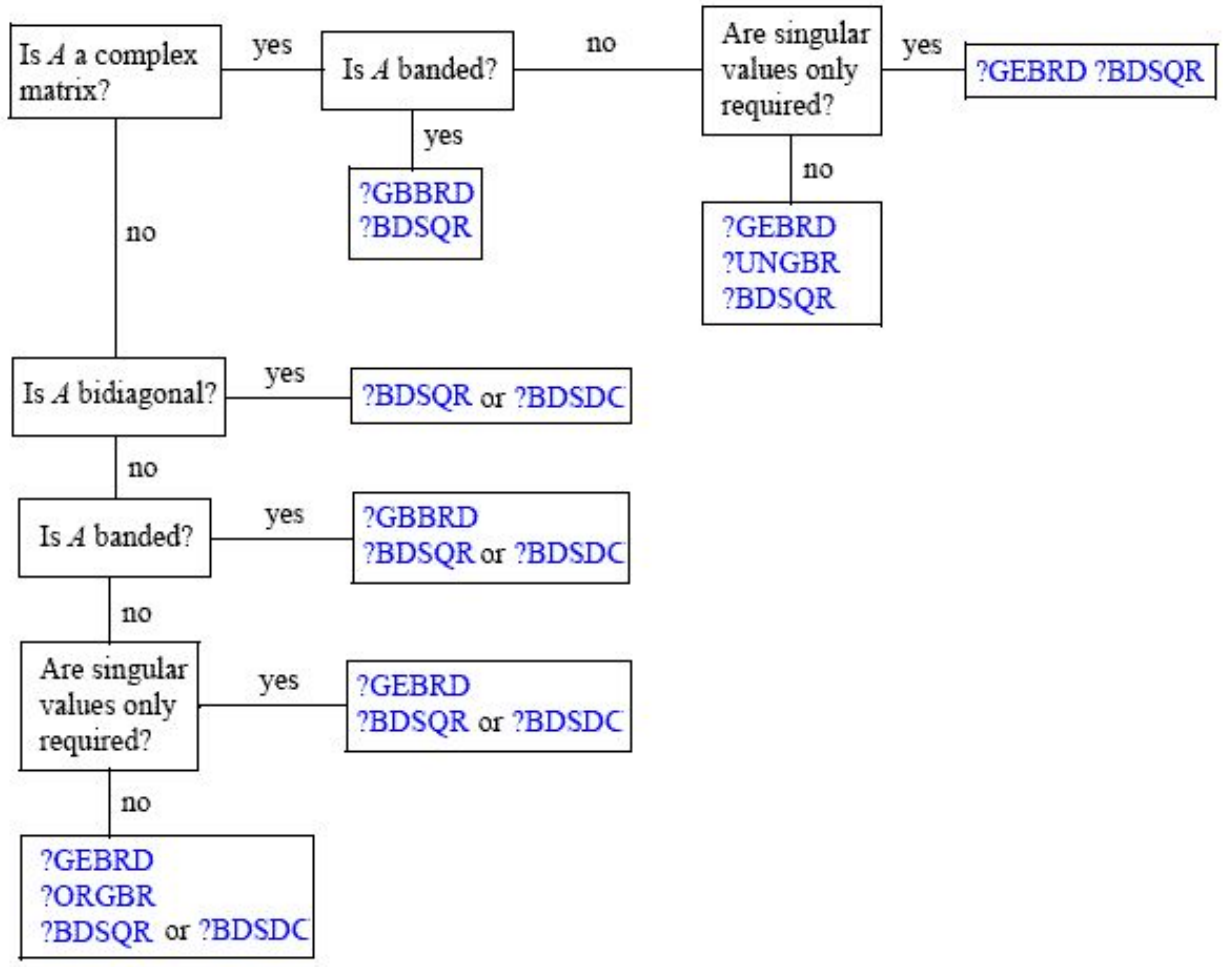


Figure "Decision Tree: Singular Value Decomposition" presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether *A* is real or complex, and so on.

You can use the SVD to find a minimum-norm solution to a (possibly) rank-deficient least squares problem of minimizing $||Ax - b||^2$. The effective rank *k* of the matrix *A* can be determined as the number of singular values which exceed a suitable threshold. The minimum-norm solution is

$$x = V_k(\Sigma_k)^{-1}c$$

where Σ_k is the leading *k*-by-*k* submatrix of Σ, the matrix V_k consists of the first *k* columns of $V = PV_1$, and the vector *c* consists of the first *k* elements of $U^Hb = U_1{}^HQ^Hb$.

?gebrd
Reduces a general matrix to bidiagonal form.

Syntax

```
call sgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call dgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call cgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call zgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
```

```
call gebrd(a [, d] [,e] [,tauq] [,taup] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a general *m*-by-*n* matrix *A* to a bidiagonal matrix *B* by an orthogonal (unitary) transformation.

If $m \geq n$, the reduction is given by $A = QBP^H = \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P_H$,

where B_1 is an *n*-by-*n* upper diagonal matrix, *Q* and *P* are orthogonal or, for a complex *A*, unitary matrices; Q_1 consists of the first *n* columns of *Q*.

If $m < n$, the reduction is given by

$A = Q*B*P^H = Q*(B_1 0)*P^H = Q_1*B_1*P_1{}^H$,

where B_1 is an *m*-by-*m* lower diagonal matrix, *Q* and *P* are orthogonal or, for a complex *A*, unitary matrices; P_1 consists of the first *m* columns of *P*.

The routine does not form the matrices *Q* and *P* explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices *Q* and *P* in this representation:

If the matrix *A* is real,

- to compute *Q* and *P* explicitly, call orgbr.
- to multiply a general matrix by *Q* or *P*, call ormbr.

If the matrix *A* is complex,

- to compute *Q* and *P* explicitly, call ungbr.
- to multiply a general matrix by *Q* or *P*, call unmbr.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq 0$).
a, *work*	REAL for sgebrd DOUBLE PRECISION for dgebrd COMPLEX for cgebrd DOUBLE COMPLEX for zgebrd. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The dimension of *work*; at least max(1, *m*, *n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a

If *m*≥*n*, the diagonal and first super-diagonal of *a* are overwritten by the upper bidiagonal matrix *B*. The elements below the diagonal, with the array *tauq*, represent the orthogonal matrix *Q* as a product of elementary reflectors, and the elements above the first superdiagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors.

If *m* < *n*, the diagonal and first sub-diagonal of *a* are overwritten by the lower bidiagonal matrix *B*. The elements below the first subdiagonal, with the array *tauq*, represent the orthogonal matrix *Q* as a product of elementary reflectors, and the elements above the diagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors.

d

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size at least max(1, min(*m*, *n*)).

Contains the diagonal elements of *B*.

e

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size at least max(1, min(*m*, *n*) - 1). Contains the off-diagonal elements of *B*.

tauq, *taup*

REAL for sgebrd

DOUBLE PRECISION for dgebrd

COMPLEX for cgebrd

DOUBLE COMPLEX for zgebrd.

Arrays, size at least max (1, min(*m*, *n*)). The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrices *P* and *Q*.

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gebrd` interface are the following:

`a`	Holds the matrix *A* of size (*m*,*n*).
`d`	Holds the vector of length min(*m*,*n*).
`e`	Holds the vector of length min(*m*,*n*)-1.
`tauq`	Holds the vector of length min(*m*,*n*).
`taup`	Holds the vector of length min(*m*,*n*).

Application Notes

For better performance, try using `lwork = (m + n)*blocksize`, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrices *Q*, *B*, and *P* satisfy $QBP^H = A + E$, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$(4/3)*n^2*(3*m - n)$ for $m \geq n$,

$(4/3)*m^2*(3*n - m)$ for $m < n$.

The number of operations for complex flavors is four times greater.

If *n* is much less than *m*, it can be more efficient to first form the *QR* factorization of *A* by calling geqrf and then reduce the factor *R* to bidiagonal form. This requires approximately $2*n^2*(m + n)$ floating-point operations.

If *m* is much less than *n*, it can be more efficient to first form the *LQ* factorization of *A* by calling gelqf and then reduce the factor *L* to bidiagonal form. This requires approximately $2*m^2*(m + n)$ floating-point operations.

?gbbrd

Reduces a general band matrix to bidiagonal form.

Syntax

```
call sgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work, info)
```

```
call dgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work,
info)
```

```
call cgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work,
rwork, info)
```

```
call zgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work,
rwork, info)
```

```
call gbbrd(ab [, c] [,d] [,e] [,q] [,pt] [,kl] [,m] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces an *m*-by-*n* band matrix *A* to upper bidiagonal matrix *B*: $A = Q*B*P^H$. Here the matrices *Q* and *P* are orthogonal (for real *A*) or unitary (for complex *A*). They are determined as products of Givens rotation matrices, and may be formed explicitly by the routine if required. The routine can also update a matrix *C* as follows: $C = Q^H*C$.

Input Parameters

vect	CHARACTER*1. Must be 'N' or 'Q' or 'P' or 'B'. If *vect* = 'N', neither *Q* nor P^H is generated. If *vect* = 'Q', the routine generates the matrix *Q*. If *vect* = 'P', the routine generates the matrix P^H. If *vect* = 'B', the routine generates both *Q* and P^H.
m	INTEGER. The number of rows in the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq 0$).
ncc	INTEGER. The number of columns in *C* ($ncc \geq 0$).
kl	INTEGER. The number of sub-diagonals within the band of *A* ($kl \geq 0$).
ku	INTEGER. The number of super-diagonals within the band of *A* ($ku \geq 0$).
ab, *c*, *work*	REAL for sgbbrd DOUBLE PRECISION for dgbbrd COMPLEX for cgbbrd DOUBLE COMPLEX for zgbbrd. Arrays: *ab*(*ldab*,*) contains the matrix *A* in band storage (see Matrix Storage Schemes). The second dimension of *a* must be at least max(1, *n*). *c*(*ldc*,*) contains an *m*-by-*ncc* matrix *C*. If *ncc* = 0, the array *c* is not referenced. The second dimension of *c* must be at least max(1, *ncc*). *work*(*) is a workspace array.

The dimension of *work* must be at least 2*max(*m*, *n*) for real flavors, or max(*m*, *n*) for complex flavors.

ldab INTEGER. The leading dimension of the array *ab* (*ldab*≥*kl* + *ku* + 1).

ldq INTEGER. The leading dimension of the output array *q*.

ldq≥ max(1, *m*) if *vect* = 'Q' or 'B', *ldq*≥ 1 otherwise.

ldpt INTEGER. The leading dimension of the output array *pt*.

ldpt≥ max(1, *n*) if *vect* = 'P' or 'B', *ldpt*≥ 1 otherwise.

ldc INTEGER. The leading dimension of the array *c*.

ldc≥ max(1, *m*) if *ncc* > 0; *ldc*≥ 1 if *ncc* = 0.

rwork REAL for cgbbrd DOUBLE PRECISION for zgbbrd.

A workspace array, size at least max(*m*, *n*).

Output Parameters

ab Overwritten by values generated during the reduction.

d REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size at least max(1, min(*m*, *n*)). Contains the diagonal elements of the matrix *B*.

e REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size at least max(1, min(*m*, *n*) – 1).

Contains the off-diagonal elements of *B*.

q, *pt* REAL for sgebrd

DOUBLE PRECISION for dgebrd

COMPLEX for cgebrd

DOUBLE COMPLEX for zgebrd.

Arrays:

q(*ldq*,*) contains the output *m*-by-*m* matrix *Q*.

The second dimension of *q* must be at least max(1, *m*).

p(*ldpt*,*) contains the output *n*-by-*n* matrix P^T.

The second dimension of *pt* must be at least max(1, *n*).

c Overwritten by the product $Q^H * C$.

c is not referenced if *ncc* = 0.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gbbrd interface are the following:

ab	Holds the array *A* of size (*kl*+*ku*+1,*n*).
c	Holds the matrix *C* of size (*m*,*ncc*).
d	Holds the vector with the number of elements min(*m*,*n*).
e	Holds the vector with the number fo elements min(*m*,*n*)-1.
q	Holds the matrix *Q* of size (*m*,*m*).
pt	Holds the matrix *PT* of size (*n*,*n*).
m	If omitted, assumed *m* = *n*.
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *ku* = *lda*-*kl*-1.
vect	Restored based on the presence of arguments *q* and pt as follows: *vect* = 'B', if both *q* and *pt* are present, *vect* = 'Q', if *q* is present and *pt* omitted, *vect* = 'P', if *q* is omitted and *pt* present, *vect* = 'N', if both *q* and *pt* are omitted.

Application Notes

The computed matrices *Q*, *B*, and *P* satisfy $Q*B*P^H = A + E$, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

If *m* = *n*, the total number of floating-point operations for real flavors is approximately the sum of:

$6*n^2*(kl + ku)$ if *vect* = 'N' and *ncc* = 0,

$3*n^2*ncc*(kl + ku - 1)/(kl + ku)$ if *C* is updated, and

$3*n^3*(kl + ku - 1)/(kl + ku)$ if either *Q* or P^H is generated (double this if both).

To estimate the number of operations for complex flavors, use the same formulas with the coefficients 20 and 10 (instead of 6 and 3).

?orgbr
Generates the real orthogonal matrix Q or P^T determined by ?gebrd.

Syntax

```
call sorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call dorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call orgbr(a, tau [,vect] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates the whole or part of the orthogonal matrices Q and P^T formed by the routines gebrd. Use this routine after a call to sgebrd/dgebrd. All valid combinations of arguments are described in *Input parameters*. In most cases you need the following:

To compute the whole m-by-m matrix Q:

```
call ?orgbr('Q', m, m, n, a ... )
```

(note that the array *a* must have at least *m* columns).

To form the n leading columns of Q if $m > n$:

```
call ?orgbr('Q', m, n, n, a ... )
```

To compute the whole n-by-n matrix P^T:

```
call ?orgbr('P', n, n, m, a ... )
```

(note that the array *a* must have at least *n* rows).

To form the m leading rows of P^T if $m < n$:

```
call ?orgbr('P', m, n, m, a ... )
```

Input Parameters

vect

CHARACTER*1. Must be 'Q' or 'P'.

If *vect* = 'Q', the routine generates the matrix Q.

If *vect* = 'P', the routine generates the matrix P^T.

m, n

INTEGER. The number of rows (m) and columns (n) in the matrix Q or P^T to be returned ($m \geq 0$, $n \geq 0$).

If *vect* = 'Q', $m \geq n \geq \min(m, k)$.

If *vect* = 'P', $n \geq m \geq \min(n, k)$.

k

If *vect* = 'Q', the number of columns in the original m-by-k matrix reduced by gebrd.

If *vect* = 'P', the number of rows in the original k-by-n matrix reduced by gebrd.

a

REAL for sorgbr

DOUBLE PRECISION for dorgbr

The vectors which define the elementary reflectors, as returned by gebrd.

lda

INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, m)$.

tau

REAL for sorgbr

DOUBLE PRECISION for dorgbr

Array, size `min (m,k) if vect = 'Q'`, `min (n,k) if vect = 'P'`. Scalar factor of the elementary reflector $H(i)$ or $G(i)$, which determines Q and P^T as returned by `gebrd` in the array *tauq* or *taup*.

work

`REAL` for `sorgbr`

`DOUBLE PRECISION` for `dorgbr`

Workspace array, size `max(1, lwork)`.

lwork

`INTEGER`. Dimension of the array *work*. See *Application Notes* for the suggested value of *lwork*.

If `lwork = -1` then the routine performs a workspace query and calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

a

Overwritten by the orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by *vect*, *m*, and *n*.

work(1)

If `info = 0`, on exit `work(1)` contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

`INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `orgbr` interface are the following:

a

Holds the matrix A of size (m,n).

tau

Holds the vector of length min(m,k) where

`k = m`, if `vect = 'P'`,

`k = n`, if `vect = 'Q'`.

vect

Must be `'Q'` or `'P'`. The default value is `'Q'`.

Application Notes

For better performance, try using `lwork = min(m,n)*blocksize`, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$.

The approximate numbers of floating-point operations for the cases listed in *Description* are as follows:

To form the whole of *Q*:

$(4/3)*n*(3m^2 - 3m*n + n^2)$ if $m > n$;

$(4/3)*m^3$ if $m \le n$.

To form the *n* leading columns of *Q* when $m > n$:

$(2/3)*n^2*(3m - n)$ if $m > n$.

To form the whole of P^T:

$(4/3)*n^3$ if $m \ge n$;

$(4/3)*m*(3n^2 - 3m*n + m^2)$ if $m < n$.

To form the *m* leading columns of P^T when $m < n$:

$(2/3)*n^2*(3m - n)$ if $m > n$.

The complex counterpart of this routine is ungbr.

?ormbr
Multiplies an arbitrary real matrix by the real orthogonal matrix Q or P^T determined by ?gebrd.

Syntax

```
call sormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call ormbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

Given an arbitrary real matrix *C*, this routine forms one of the matrix products $Q*C$, Q^T*C, $C*Q$, $C*Q^T$, $P*C$, P^T*C, $C*P$, $C*P^T$, where *Q* and *P* are orthogonal matrices computed by a call to gebrd. The routine overwrites the product on *C*.

Input Parameters

In the descriptions below, *r* denotes the order of *Q* or P^T:

If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

vect CHARACTER*1. Must be 'Q' or 'P'.
If *vect* = 'Q', then *Q* or Q^T is applied to *C*.

If *vect* = 'P', then P or P_T is applied to C.

side — CHARACTER*1. Must be 'L' or 'R'.

If *side* = 'L', multipliers are applied to C from the left.

If *side* = 'R', they are applied to C from the right.

trans — CHARACTER*1. Must be 'N' or 'T'.

If *trans* = 'N', then Q or P is applied to C.

If *trans* = 'T', then Q^T or P^T is applied to C.

m — INTEGER. The number of rows in C.

n — INTEGER. The number of columns in C.

k — INTEGER. One of the dimensions of A in ?gebrd:

If *vect* = 'Q', the number of columns in A;

If *vect* = 'P', the number of rows in A.

Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.

a, *c*, *work* — REAL for sormbr

DOUBLE PRECISION for dormbr.

Arrays:

a(*lda*,*) is the array *a* as returned by ?gebrd.

Its second dimension must be at least max(1, min(*r*,*k*)) for *vect* = 'Q', or max(1, *r*)) for *vect* = 'P'.

c(*ldc*,*) holds the matrix C.

Its second dimension must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*. Constraints:

lda ≥ max(1, *r*) if *vect* = 'Q';

lda ≥ max(1, min(*r*,*k*)) if *vect* = 'P'.

ldc — INTEGER. The leading dimension of *c*; *ldc* ≥ max(1, *m*) .

tau — REAL for sormbr

DOUBLE PRECISION for dormbr.

Array, size at least max (1, min(*r*, *k*)).

For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd.

lwork — INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product $Q*C$, Q^T*C, $C*Q$, $C*Q,^T$, $P*C$, P^T*C, $C*P$, or $C*P^T$, as specified by *vect*, *side*, and *trans*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ormbr interface are the following:

a — Holds the matrix *A* of size (*r*,min*(nq,k)*) where

r = *nq*, if *vect* = 'Q',

r = min(*nq*,*k*), if *vect* = 'P',

nq = *m*, if *side* = 'L',

nq = *n*, if *side* = 'R',

k = *m*, if *vect* = 'P',

k = *n*, if *vect* = 'Q'.

tau — Holds the vector of length min(*nq*,*k*).

c — Holds the matrix *C* of size (*m*,*n*).

vect — Must be 'Q' or 'P'. The default value is 'Q'.

side — Must be 'L' or 'R'. The default value is 'L'.

trans — Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using

lwork = *n***blocksize* for *side* = 'L', or

lwork = *m***blocksize* for *side* = 'R',

where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $||E||_2 = O(\varepsilon)*||C||_2$.

The total number of floating-point operations is approximately

$2*n*k(2*m - k)$ if `side = 'L'` and $m \geq k$;

$2*m*k(2*n - k)$ if `side = 'R'` and $n \geq k$;

$2*m^2*n$ if `side = 'L'` and $m < k$;

$2*n^2*m$ if `side = 'R'` and $n < k$.

The complex counterpart of this routine is unmbr.

?ungbr
Generates the complex unitary matrix Q or P^H determined by `?gebrd`.

Syntax

```
call cungbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call zungbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call ungbr(a, tau [,vect] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine generates the whole or part of the unitary matrices Q and P^H formed by the routines gebrd. Use this routine after a call to `cgebrd`/`zgebrd`. All valid combinations of arguments are described in *Input Parameters*; in most cases you need the following:

To compute the whole m-by-m matrix Q, use:

```
call ?ungbr('Q', m, m, n, a ... )
```

(note that the array*a* must have at least m columns).

To form the n leading columns of Q if $m > n$, use:

```
call ?ungbr('Q', m, n, n, a ... )
```

To compute the whole n-by-n matrix P^H, use:

```
call ?ungbr('P', n, n, m, a ... )
```

(note that the array *a* must have at least n rows).

To form the *m* leading rows of P^H if $m < n$, use:

```
call ?ungbr('P', m, n, m, a ... )
```

Input Parameters

vect — CHARACTER*1. Must be 'Q' or 'P'.

If *vect* = 'Q', the routine generates the matrix *Q*.

If *vect* = 'P', the routine generates the matrix P^H.

m — INTEGER. The number of required rows of *Q* or P^H.

n — INTEGER. The number of required columns of *Q* or P^H.

k — INTEGER. One of the dimensions of *A* in ?gebrd:

If *vect* = 'Q', the number of columns in *A*;

If *vect* = 'P', the number of rows in *A*.

Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.

For *vect* = 'Q': $k \leq n \leq m$ if $m > k$, or $m = n$ if $m \leq k$.

For *vect* = 'P': $k \leq m \leq n$ if $n > k$, or $m = n$ if $n \leq k$.

a, *work* — COMPLEX for cungbr

DOUBLE COMPLEX for zungbr.

Arrays:

a(*lda*,*) is the array *a* as returned by ?gebrd.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; at least max(1, *m*).

tau — COMPLEX for cungbr

DOUBLE COMPLEX for zungbr.

For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd.

The dimension of *tau* must be at least max(1, min(*m*, *k*)) for *vect* = 'Q', or max(1, min(*m*, *k*)) for *vect* = 'P'.

lwork — INTEGER. The size of the *work* array.

Constraint: *lwork* < max(1, min(*m*, *n*)).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by *vect*, *m*, and *n*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ungbr interface are the following:

a	Holds the matrix A of size (m,n).
tau	Holds the vector of length min(m,k) where $k = m$, if *vect* = 'P', $k = n$, if *vect* = 'Q'.
vect	Must be 'Q' or 'P'. The default value is 'Q'.

Application Notes

For better performance, try using *lwork* = min(*m*,*n*)***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $||E||_2 = O(\varepsilon)$.

The approximate numbers of possible floating-point operations are listed below:

To compute the whole matrix Q:

$(16/3)n(3m^2 - 3m*n + n^2)$ if $m > n$;

$(16/3)m^3$ if $m \le n$.

To form the n leading columns of Q when $m > n$:

$(8/3)n^2(3m - n^2)$.

To compute the whole matrix P^H:

(16/3)n^3 if $m \geq n$;

(16/3)$m(3n^2 - 3m*n + m^2)$ if $m < n$.

To form the m leading columns of P^H when $m < n$:

(8/3)$n^2(3m - n^2)$ if $m > n$.

The real counterpart of this routine is orgbr.

?unmbr

Multiplies an arbitrary complex matrix by the unitary matrix Q or P determined by ?gebrd.

Syntax

```
call cunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call unmbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

Given an arbitrary complex matrix C, this routine forms one of the matrix products $Q*C$, Q^H*C, $C*Q$, $C*Q^H$, $P*C$, P^H*C, $C*P$, or $C*P^H$, where Q and P are unitary matrices computed by a call to gebrd/gebrd. The routine overwrites the product on C.

Input Parameters

In the descriptions below, r denotes the order of Q or P^H:

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

vect
CHARACTER*1. Must be 'Q' or 'P'.
If *vect* = 'Q', then Q or Q^H is applied to C.
If *vect* = 'P', then P or P^H is applied to C.

side
CHARACTER*1. Must be 'L' or 'R'.
If *side* = 'L', multipliers are applied to C from the left.
If *side* = 'R', they are applied to C from the right.

trans
CHARACTER*1. Must be 'N' or 'C'.
If *trans* = 'N', then Q or P is applied to C.
If *trans* = 'C', then Q^H or P^H is applied to C.

m
INTEGER. The number of rows in C.

n
INTEGER. The number of columns in C.

k
INTEGER. One of the dimensions of A in ?gebrd:
If *vect* = 'Q', the number of columns in A;

If *vect* = 'P', the number of rows in *A*.

Constraints: *m*≥ 0, *n*≥ 0, *k*≥ 0.

a, *c*, *work*

COMPLEX for cunmbr

DOUBLE COMPLEX for zunmbr.

Arrays:

a(*lda*,*) is the array *a* as returned by ?gebrd.

Its second dimension must be at least max(1, min(*r*,*k*)) for *vect* = 'Q', or max(1, *r*)) for *vect* = 'P'.

c(*ldc*,*) holds the matrix *C*.

Its second dimension must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of *a*. Constraints:

lda≥ max(1, *r*) if *vect* = 'Q';

lda≥ max(1, min(*r*,*k*)) if *vect* = 'P'.

ldc

INTEGER. The leading dimension of *c*; *ldc*≥ max(1, *m*).

tau

COMPLEX for cunmbr

DOUBLE COMPLEX for zunmbr.

Array, size at least max (1, min(*r*, *k*)).

For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd.

lwork

INTEGER. The size of the *work* array.

lwork≥ max(1, *n*) if *side* = 'L';

lwork≥ max(1, *m*) if *side* = 'R'.

lwork≥ 1 if *n*=0 or *m*=0.

For optimum performance *lwork* ≥ max(1,*n***nb*) if *side* = 'L', and *lwork* ≥ max(1,*m***nb*) if *side* = 'R', where *nb* is the optimal blocksize. (*nb* = 0 if *m* = 0 or *n* = 0.)

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c

Overwritten by the product *Q***C*, Q^H**C*, *C***Q*, *C**Q^H, *P***C*, P^H**C*, *C***P*, or *C**P^H, as specified by *vect*, *side*, and *trans*.

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine unmbr interface are the following:

a	Holds the matrix *A* of size (*r*,min*(nq,k)*) where *r* = *nq*, if *vect* = 'Q', *r* = min(*nq*,*k*), if *vect* = 'P', *nq* = *m*, if *side* = 'L', *nq* = *n*, if *side* = 'R', *k* = *m*, if *vect* = 'P', *k* = *n*, if *vect* = 'Q'.
tau	Holds the vector of length min(*nq,k*).
c	Holds the matrix *C* of size (*m,n*).
vect	Must be 'Q' or 'P'. The default value is 'Q'.
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, use

lwork = *n***blocksize* for *side* = 'L', or

lwork = *m***blocksize* for *side* = 'R',

where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $||E||_2 = O(\varepsilon)*||C||_2$.

The total number of floating-point operations is approximately

`8*n*k(2*m - k)` if `side = 'L'` and $m \geq k$;

`8*m*k(2*n - k)` if `side = 'R'` and $n \geq k$;

$8*m^2*n$ if `side = 'L'` and $m < k$;

$8*n^2*m$ if `side = 'R'` and $n < k$.

The real counterpart of this routine is ormbr.

?bdsqr

Computes the singular value decomposition of a general matrix that has been reduced to bidiagonal form.

Syntax

```
call sbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
call dbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
call cbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, rwork, info)
call zbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, rwork, info)
call rbdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
call bdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the singular values and, optionally, the right and/or left singular vectors from the Singular Value Decomposition (SVD) of a real *n*-by-*n* (upper or lower) bidiagonal matrix *B* using the implicit zero-shift *QR* algorithm. The SVD of *B* has the form $B = Q*S*P^H$ where *S* is the diagonal matrix of singular values, *Q* is an orthogonal matrix of left singular vectors, and *P* is an orthogonal matrix of right singular vectors. If left singular vectors are requested, this subroutine actually returns *U* **Q* instead of *Q*, and, if right singular vectors are requested, this subroutine returns P^H **VT* instead of P^H, for given real/complex input matrices *U* and *VT*. When *U* and *VT* are the orthogonal/unitary matrices that reduce a general matrix *A* to bidiagonal form: `A = U*B*VT`, as computed by `?gebrd`, then

$A = (U*Q)*S*(P^H*VT)$

is the SVD of *A*. Optionally, the subroutine may also compute Q^H **C* for a given real/complex input matrix *C*.

See also lasq1, lasq2, lasq3, lasq4, lasq5, lasq6 used by this routine.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If `uplo = 'U'`, *B* is an upper bidiagonal matrix. If `uplo = 'L'`, *B* is a lower bidiagonal matrix.
n	INTEGER. The order of the matrix *B* ($n \geq 0$).
ncvt	INTEGER. The number of columns of the matrix *VT*, that is, the number of right singular vectors ($ncvt \geq 0$).

Set *ncvt* = 0 if no right singular vectors are required.

nru — INTEGER. The number of rows in *U*, that is, the number of left singular vectors (*nru*≥ 0).

Set *nru* = 0 if no left singular vectors are required.

ncc — INTEGER. The number of columns in the matrix *C* used for computing the product Q^H*C (*ncc*≥ 0). Set *ncc* = 0 if no matrix *C* is supplied.

d, *e* — REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays:

d(*) contains the diagonal elements of *B*.

The size of *d* must be at least max(1, *n*).

e(*) contains the (*n*–1) off-diagonal elements of *B*.

The size of *e* must be at least max(1, *n* – 1).

work — REAL for sbdsqr

DOUBLE PRECISION for dbdsqr.

work(*) is a workspace array.

The size of *work* must be at least max(1, 4**n*).

rwork — REAL for cbdsqr

DOUBLE PRECISION for zbdsqr.

rwork(*) is a workspace array.

The size of *rwork* must be at least max(1, 4**n*).

vt, *u*, *c* — REAL for sbdsqr

DOUBLE PRECISION for dbdsqr

COMPLEX for cbdsqr

DOUBLE COMPLEX for zbdsqr.

Arrays:

vt(*ldvt*,*) contains an *n*-by-*ncvt* matrix *VT*.

The second dimension of *vt* must be at least max(1, *ncvt*).

vt is not referenced if *ncvt* = 0.

u(*ldu*,*) contains an *nru* by *n* matrix *U*.

The second dimension of *u* must be at least max(1, *n*).

u is not referenced if *nru* = 0.

c(*ldc*,*) contains the *n*-by-*ncc* matrix *C* for computing the product Q^H*C.

The second dimension of *c* must be at least max(1, *ncc*). The array is not referenced if *ncc* = 0.

ldvt — INTEGER. The leading dimension of *vt*. Constraints:

ldvt ≥ max(1, *n*) if *ncvt* > 0;

ldvt ≥ 1 if *ncvt* = 0.

ldu INTEGER. The leading dimension of *u*. Constraint:

ldu ≥ max(1, *nru*) .

ldc INTEGER. The leading dimension of *c*. Constraints:

ldc ≥ max(1, *n*) if *ncc* > 0; *ldc* ≥ 1 otherwise.

Output Parameters

d On exit, if *info* = 0, overwritten by the singular values in decreasing order (see *info*).

e On exit, if *info* = 0, *e* is destroyed. See also *info* below.

c Overwritten by the product Q^H*C*.

vt On exit, this array is overwritten by P^H *VT*. Not referenced if *ncvt* = 0.

u On exit, this array is overwritten by *U* *Q. Not referenced if *nru* = 0.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0,

If *ncvt* = *nru* = *ncc* = 0,

- *info* = 1, a split was marked by a positive value in *e*
- *info* = 2, the current block of *z* not diagonalized after 100**n* iterations (in the inner `while` loop)
- *info* = 3, termination criterion of the outer `while` loop is not met (the program created more than *n* unreduced blocks).

In all other cases when *ncvt*, *nru*, or *ncc* > 0, the algorithm did not converge; *d* and *e* contain the elements of a bidiagonal matrix that is orthogonally similar to the input matrix *B*; if *info* = *i*, *i* elements of *e* have not converged to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `bdsqr` interface are the following:

d Holds the vector of length (*n*).

e Holds the vector of length (*n*).

vt Holds the matrix *VT* of size (*n*, *ncvt*).

u Holds the matrix *U* of size (*nru*,*n*).

c	Holds the matrix *C* of size (*n*,*ncc*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
ncvt	If argument *vt* is present, then *ncvt* is equal to the number of columns in matrix *VT*; otherwise, *ncvt* is set to zero.
nru	If argument *u* is present, then *nru* is equal to the number of rows in matrix *U*; otherwise, *nru* is set to zero.
ncc	If argument *c* is present, then *ncc* is equal to the number of columns in matrix *C*; otherwise, *ncc* is set to zero.

Note that two variants of Fortran 95 interface for bdsqr routine are needed because of an ambiguous choice between real and complex cases appear when *vt*, *u*, and *c* are omitted. Thus, the name rbdsqr is used in real cases (single or double precision), and the name bdsqr is used in complex cases (single or double precision).

Application Notes

Each singular value and singular vector is computed to high relative accuracy. However, the reduction to bidiagonal form (prior to calling the routine) may decrease the relative accuracy in the small singular values of the original matrix if its singular values vary widely in magnitude.

If s_i is an exact singular value of *B*, and s_i is the corresponding computed value, then

$|s_i - \sigma_i| \leq p(m,n) * \varepsilon * \sigma_i$

where $p(m, n)$ is a modestly increasing function of *m* and *n*, and ε is the machine precision.

If only singular values are computed, they are computed more accurately than when some singular vectors are also computed (that is, the function $p(m, n)$ is smaller).

If u_i is the corresponding exact left singular vector of *B*, and w_i is the corresponding computed left singular vector, then the angle $\theta(u_i, w_i)$ between them is bounded as follows:

$\theta(u_i, w_i) \leq p(m,n) * \varepsilon / \min_{i \neq j}(|\sigma i - \sigma j| / |\sigma_i + \sigma j|)$.

Here $\min_{i \neq j}(|\sigma_i - \sigma_j| / |\sigma_i + \sigma_j|)$ is the *relative gap* between σ_i and the other singular values. A similar error bound holds for the right singular vectors.

The total number of real floating-point operations is roughly proportional to n^2 if only the singular values are computed. About $6n^2 * nru$ additional operations ($12n^2 * nru$ for complex flavors) are required to compute the left singular vectors and about $6n^2 * ncvt$ operations ($12n^2 * ncvt$ for complex flavors) to compute the right singular vectors.

?bdsdc

Computes the singular value decomposition of a real bidiagonal matrix using a divide and conquer method.

Syntax

```
call sbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
call dbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
call bdsdc(d, e [,u] [,vt] [,q] [,iq] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the Singular Value Decomposition (SVD) of a real *n*-by-*n* (upper or lower) bidiagonal matrix *B*: $B = U*\Sigma*V^T$, using a divide and conquer method, where Σ is a diagonal matrix with non-negative diagonal elements (the singular values of *B*), and *U* and *V* are orthogonal matrices of left and right singular vectors, respectively. ?bdsdc can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

This rotuine
uses ?lasd0, ?lasd1, ?lasd2, ?lasd3, ?lasd4, ?lasd5, ?lasd6, ?lasd7, ?lasd8, ?lasd9, ?lasda, ?lasdq, ?lasdt.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *B* is an upper bidiagonal matrix.

If *uplo* = 'L', *B* is a lower bidiagonal matrix.

compq

CHARACTER*1. Must be 'N', 'P', or 'I'.

If *compq* = 'N', compute singular values only.

If *compq* = 'P', compute singular values and compute singular vectors in compact form.

If *compq* = 'I', compute singular values and singular vectors.

n

INTEGER. The order of the matrix *B* ($n \geq 0$).

d, *e*, *work*

REAL for sbdsdc

DOUBLE PRECISION for dbdsdc.

Arrays:

d(*) contains the *n* diagonal elements of the bidiagonal matrix *B*. The size of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of the bidiagonal matrix *B*. The size of *e* must be at least max(1, *n*).

work(*) is a workspace array.

The dimension of *work* must be at least:

max(1, 4**n*), if *compq* = 'N' or *compq* = 'P';

max(1, $3*n^2+4*n$), if *compq* = 'I'.

ldu

INTEGER. The leading dimension of the output array *u*; *ldu*≥ 1.

If singular vectors are desired, then *ldu*≥ max(1, *n*).

ldvt

INTEGER. The leading dimension of the output array *vt*; *ldvt*≥ 1.

If singular vectors are desired, then *ldvt*≥ max(1, *n*).

iwork

INTEGER. Workspace array, dimension at least max(1, 8**n*).

Output Parameters

d

If *info* = 0, overwritten by the singular values of *B*.

e	On exit, *e* is overwritten.
u, *vt*, *q*	REAL for sbdsdc DOUBLE PRECISION for dbdsdc. Arrays: *u*(*ldu*,**), *vt*(*ldvt*,**), *q*(*). If *compq* = 'I', then on exit *u* contains the left singular vectors of the bidiagonal matrix *B*, unless *info*≠ 0 (*seeinfo*). For other values of *compq*, *u* is not referenced. The second dimension of *u* must be at least max(1,*n*). if *compq* = 'I', then on exit vt^T contains the right singular vectors of the bidiagonal matrix *B*, unless *info*≠ 0 (*seeinfo*). For other values of *compq*, *vt* is not referenced. The second dimension of *vt* must be at least max(1,*n*). If *compq* = 'P', then on exit, if *info* = 0, *q* and *iq* contain the left and right singular vectors in a compact form. Specifically, *q* contains all the REAL (for sbdsdc) or DOUBLE PRECISION (for dbdsdc) data for singular vectors. For other values of *compq*, *q* is not referenced.
iq	INTEGER. Array: *iq*(*). If *compq* = 'P', then on exit, if *info* = 0, *q* and *iq* contain the left and right singular vectors in a compact form. Specifically, *iq* contains all the INTEGER data for singular vectors. For other values of *compq*, *iq* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the algorithm failed to compute a singular value. The update process of divide and conquer failed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine bdsdc interface are the following:

d	Holds the vector of length *n*.
e	Holds the vector of length *n*.
u	Holds the matrix *U* of size (*n*,*n*).
vt	Holds the matrix *VT* of size (*n*,*n*).
q	Holds the vector of length (*ldq*), where *ldq*≥ *n**(11 + 2**smlsiz* + 8**int*(*log_2*(*n*/(*smlsiz* + 1)))) and *smlsiz* is returned by *ilaenv* and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25).

compq Restored based on the presence of arguments *u*, *vt*, *q*, and *iq* as follows:

compq = 'N', if none of *u*, *vt*, *q*, and *iq* are present,

compq = 'I', if both *u* and *vt* are present. Arguments *u* and *vt* must either be both present or both omitted,

compq = 'P', if both *q* and *iq* are present. Arguments *q* and *iq* must either be both present or both omitted.

Note that there will be an error condition if all of *u*, *vt*, *q*, and *iq* arguments are present simultaneously.

See Also

?lasd0
?lasd1
?lasd2
?lasd3
?lasd4
?lasd5
?lasd6
?lasd7
?lasd8
?lasd9
?lasda
?lasdq
?lasdt

Symmetric Eigenvalue Problems: LAPACK Computational Routines

Symmetric eigenvalue problems are posed as follows: given an *n*-by-*n* real symmetric or complex Hermitian matrix *A*, find the *eigenvalues* λ and the corresponding *eigenvectors z* that satisfy the equation

$Az = \lambda z$ (or, equivalently, $z^H A = \lambda z^H$).

In such eigenvalue problems, all *n* eigenvalues are real not only for real symmetric but also for complex Hermitian matrices *A*, and there exists an orthonormal system of *n* eigenvectors. If *A* is a symmetric or Hermitian positive-definite matrix, all eigenvalues are positive.

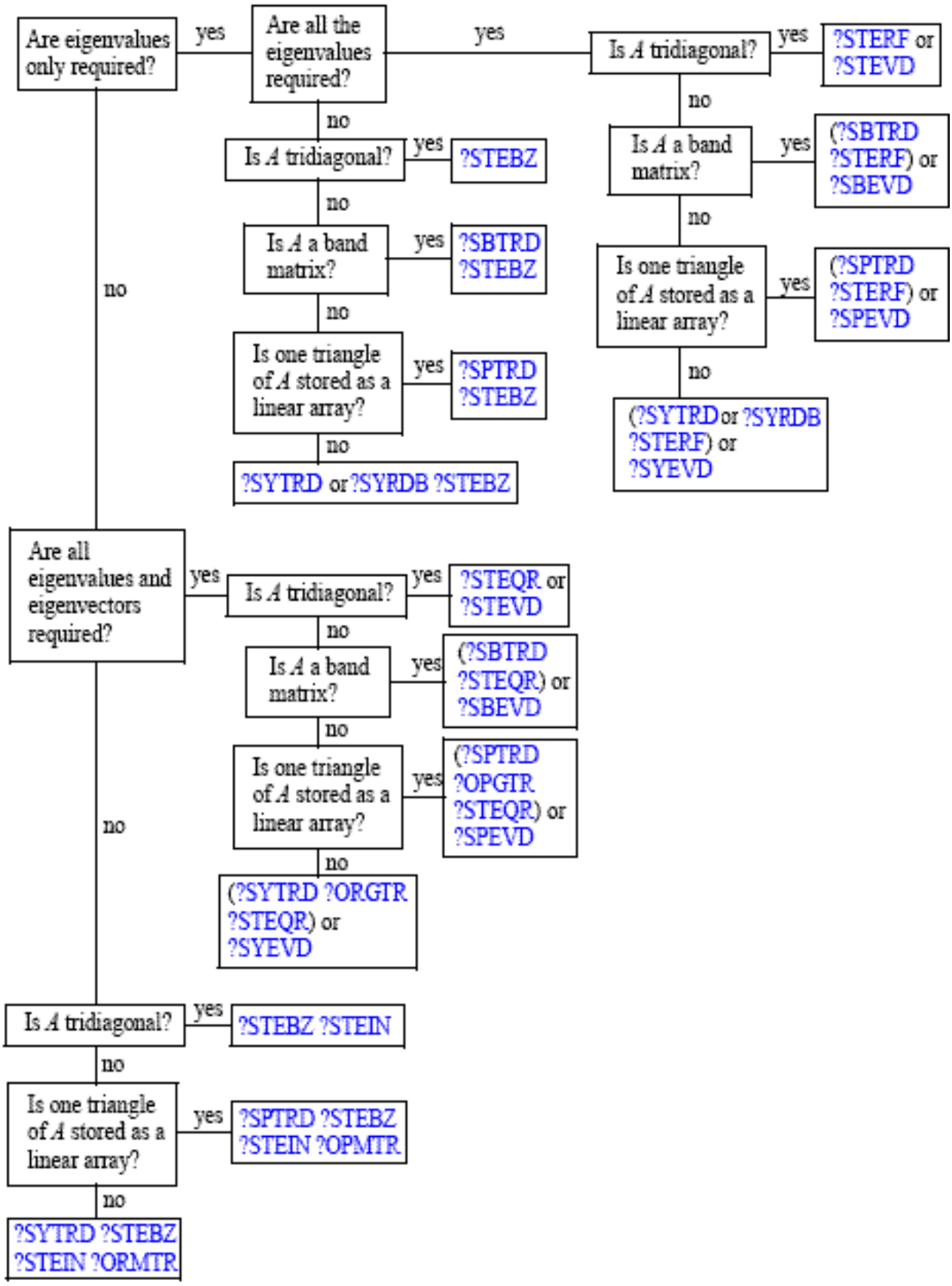
To solve a symmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to tridiagonal form and then solve the eigenvalue problem with the tridiagonal matrix obtained. LAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines (for FORTRAN 77 interface) are listed in Table "Computational Routines for Solving Symmetric Eigenvalue Problems". The corresponding routine names in the Fortran 95 interface are without the first symbol.

There are different routines for symmetric eigenvalue problems, depending on whether you need all eigenvectors or only some of them or eigenvalues only, whether the matrix *A* is positive-definite or not, and so on.

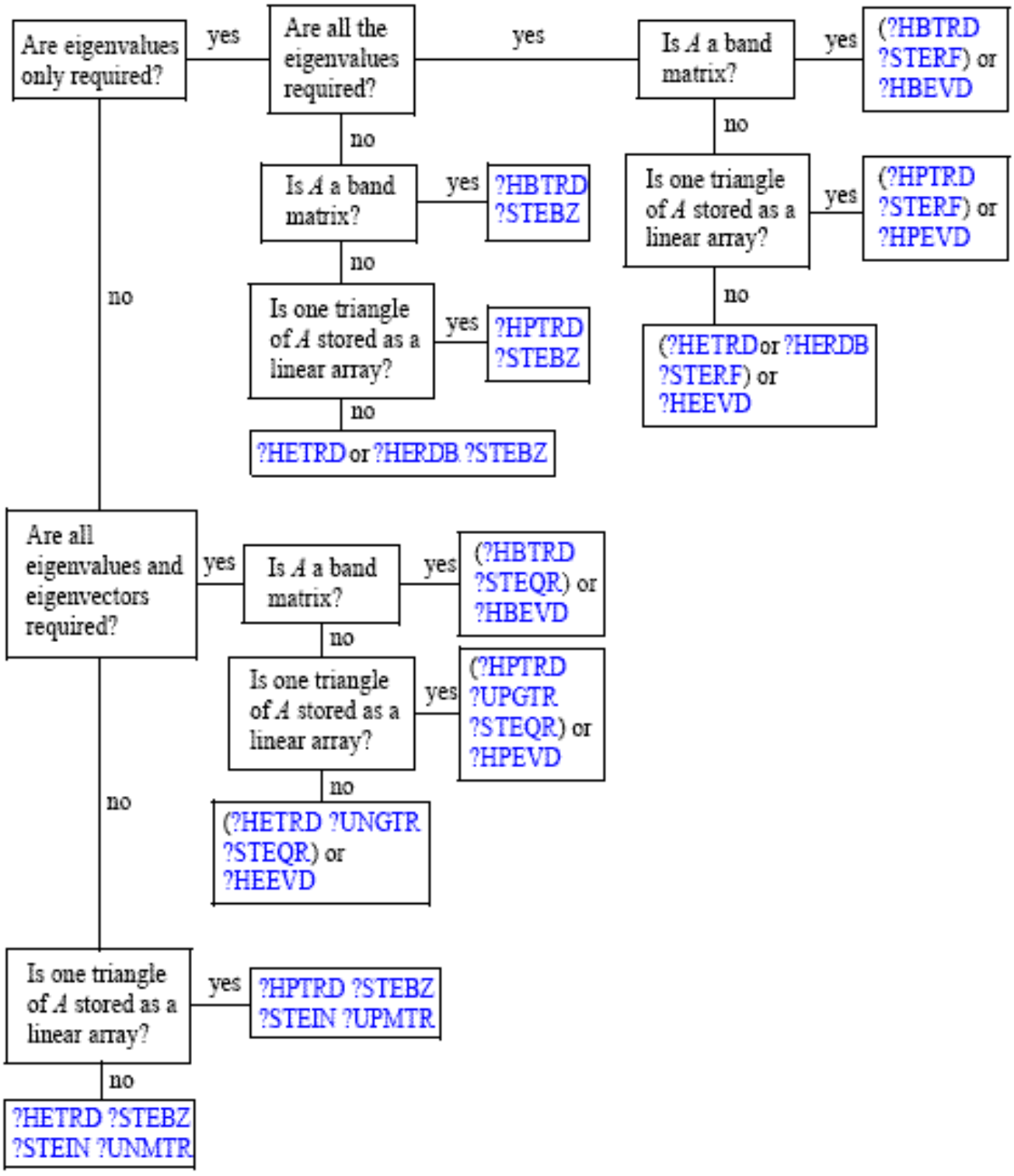
These routines are based on three primary algorithms for computing eigenvalues and eigenvectors of symmetric problems: the divide and conquer algorithm, the QR algorithm, and bisection followed by inverse iteration. The divide and conquer algorithm is generally more efficient and is recommended for computing all eigenvalues and eigenvectors. Furthermore, to solve an eigenvalue problem using the divide and conquer algorithm, you need to call only one routine. In general, more than one routine has to be called if the QR algorithm or bisection followed by inverse iteration is used.

The decision tree in Figure "Decision Tree: Real Symmetric Eigenvalue Problems" will help you choose the right routine or sequence of routines for eigenvalue problems with real symmetric matrices. Figure "Decision Tree: Complex Hermitian Eigenvalue Problems" presents a similar decision tree for complex Hermitian matrices.

Decision Tree: Real Symmetric Eigenvalue Problems



Decision Tree: Complex Hermitian Eigenvalue Problems



Computational Routines for Solving Symmetric Eigenvalue Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (full storage)	sytrd syrdb	hetrd herdb
Reduce to tridiagonal form $A = QTQ^H$ (packed storage)	sptrd	hptrd
Reduce to tridiagonal form $A = QTQ^H$ (band storage).	sbtrd	hbtrd
Generate matrix *Q* (full storage)	orgtr	ungtr
Generate matrix *Q* (packed storage)	opgtr	upgtr
Apply matrix *Q* (full storage)	ormtr	unmtr
Multiplies a general matrix by an orthogonal/unitary matrix with a 2x2 structure.	orm22	unm22
Apply matrix *Q* (packed storage)	opmtr	upmtr
Find all eigenvalues of a tridiagonal matrix *T*	sterf	
Find all eigenvalues and eigenvectors of a tridiagonal matrix *T*	steqr stedc	steqr stedc
Find all eigenvalues and eigenvectors of a tridiagonal positive-definite matrix *T*.	pteqr	pteqr
Find selected eigenvalues of a tridiagonal matrix *T*	stebz stegr	stegr
Find selected eigenvectors of a tridiagonal matrix *T*	stein stegr	stein stegr
Find selected eigenvalues and eigenvectors of f a real symmetric tridiagonal matrix *T*	stemr	stemr
Compute the reciprocal condition numbers for the eigenvectors	disna	disna

?sytrd
Reduces a real symmetric matrix to tridiagonal form.

Syntax

```
call ssytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call dsytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call sytrd(a, tau [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a real symmetric matrix *A* to symmetric tridiagonal form *T* by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix *Q* is not formed explicitly but is represented as a product of *n*-1 elementary reflectors. Routines are provided for working with *Q* in this representation (see *Application Notes* below).

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

a, *work*

REAL for ssytrd

DOUBLE PRECISION for dsytrd.

a(*lda*,*) is an array containing either upper or lower triangular part of the matrix *A*, as specified by *uplo*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of *a*; at least max(1, *n*).

lwork

INTEGER. The size of the *work* array ($lwork \geq n$).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a

On exit,

if *uplo* = 'U', the diagonal and first superdiagonal of *A* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors;

if *uplo* = 'L', the diagonal and first subdiagonal of *A* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors.

d, *e*, *tau*

REAL for ssytrd

DOUBLE PRECISION for dsytrd.

Arrays:

d(*) contains the diagonal elements of the matrix *T*.

The size of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.

The size of *e* must be at least max(1, *n*-1).

tau(*) stores (*n*-1) scalars that define elementary reflectors in decomposition of the orthogonal matrix *Q* in a product of *n*-1 elementary reflectors. *tau*(*n*) is used as workspace.

The size of *tau* must be at least max(1, *n*).

work(1) — If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sytrd interface are the following:

a — Holds the matrix *A* of size (*n*,*n*).

tau — Holds the vector of length (*n*-1).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Note that diagonal (*d*) and off-diagonal (*e*) elements of the matrix *T* are omitted because they are kept in the matrix *A* on exit.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *T* is exactly similar to a matrix $A+E$, where $||E||_2 = c(n)*\varepsilon*||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

orgtr — to form the computed matrix *Q* explicitly

ormtr to multiply a real matrix by *Q*.

The complex counterpart of this routine is ?hetrd.

?syrdb

Reduces a real symmetric matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

```
call ssyrdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
call dsyrdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Include Files

- mkl.fi

Description

The routine reduces a real symmetric matrix *A* to symmetric tridiagonal form *T* by an orthogonal similarity transformation: $A = Q*T*Q^T$ and optionally multiplies matrix *Z* by *Q*, or simply forms *Q*.

This routine reduces a full symmetric matrix *A* to the banded symmetric matrix *B*, and then to the tridiagonal symmetric matrix *T* with a Successive Bandwidth Reduction approach after C. Bischof's works (see for instance, [Bischof00]). ?syrdb is functionally close to ?sytrd routine but the tridiagonal form may differ from those obtained by ?sytrd. Unlike ?sytrd, the orthogonal matrix *Q* cannot be restored from the details of matrix *A* on exit.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only *A* is reduced to *T*.

If *jobz* = 'V', then *A* is reduced to *T* and *A* contains *Q* on exit.

If *jobz* = 'U', then *A* is reduced to *T* and *Z* contains *Z*Q* on exit.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

kd

INTEGER. The bandwidth of the banded matrix *B* ($kd \geq 1$, $kd \leq n-1$).

a,*z*, *work*

REAL for ssyrdb.

DOUBLE PRECISION for dsyrdb.

a(*lda*,*) is an array containing either upper or lower triangular part of the matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

z(*ldz*,*), the second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'U', then the matrix *z* is multiplied by *Q*.

If *jobz* = 'N' or 'V', then *z* is not referenced.

work(*lwork*) is a workspace array.

lda — INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldz — INTEGER. The leading dimension of *z*; at least max(1, *n*). Not referenced if *jobz* = 'N'

lwork — INTEGER. The size of the *work* array (*lwork*≥ (2*kd*+1)*n*+*kd*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a

If *jobz* = 'V', then overwritten by *Q* matrix.

If *jobz* = 'N' or 'U', then overwritten by the banded matrix *B* and details of the orthogonal matrix Q_B to reduce *A* to *B* as specified by *uplo*.

z

On exit,

if *jobz* = 'U', then the matrix *z* is overwritten by *Z***Q*.

If *jobz* = 'N' or 'V', then *z* is not referenced.

d, *e*, *tau* — DOUBLE PRECISION.

Arrays:

d(*) contains the diagonal elements of the matrix *T*.

The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.

The dimension of *e* must be at least max(1, *n*-1).

tau(*) stores further details of the orthogonal matrix *Q*.

The dimension of *tau* must be at least max(1, *n*-*kd*-1).

work(1) — If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n**(3**kd*+3).

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using *kd* equal to 40 if n ≤ 2000 and 64 otherwise.

Try using ?syrdb instead of ?sytrd on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. ?syrdb becomes faster beginning approximately with n = 1000, and much faster at larger matrices with a better scalability than ?sytrd.

Avoid applying ?syrdb for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to *Z* is doubled compared to the traditional one-step reduction. In that case it is better to apply ?sytrd and ?ormtr/?orgtr to obtain tridiagonal form along with the orthogonal transformation matrix *Q*.

?herdb
Reduces a complex Hermitian matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

```
call cherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
call zherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Include Files

- mkl.fi

Description

The routine reduces a complex Hermitian matrix *A* to symmetric tridiagonal form *T* by a unitary similarity transformation: $A = Q*T*Q^T$ and optionally multiplies matrix *Z* by *Q*, or simply forms *Q*.

This routine reduces a full symmetric matrix *A* to the banded symmetric matrix *B*, and then to the tridiagonal symmetric matrix *T* with a Successive Bandwidth Reduction approach after C. Bischof's works (see for instance, [Bischof00]). ?herdb is functionally close to ?hetrd routine but the tridiagonal form may differ from those obtained by ?hetrd. Unlike ?hetrd, the orthogonal matrix *Q* cannot be restored from the details of matrix *A* on exit.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only *A* is reduced to *T*.

If *jobz* = 'V', then *A* is reduced to *T* and *A* contains *Q* on exit.

If *jobz* = 'U', then *A* is reduced to *T* and *Z* contains *Z*Q* on exit.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

kd — INTEGER. The bandwidth of the banded matrix *B* (*kd* ≥ 1, *kd* ≤ *n*-1).

a, *z*, *work* — COMPLEX for cherdb.

DOUBLE COMPLEX for zherdb.

a(*lda*,*) is an array containing either upper or lower triangular part of the matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

z(*ldz*,*), the second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'U', then the matrix *z* is multiplied by *Q*.

If *jobz* = 'N' or 'V', then *z* is not referenced.

work(*lwork*) is a workspace array.

lda — INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldz — INTEGER. The leading dimension of *z*; at least max(1, *n*). Not referenced if *jobz* = 'N'

lwork — INTEGER. The size of the *work* array (*lwork* ≥ (2*kd*+1)*n*+*kd*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a

If *jobz* = 'V', then overwritten by *Q* matrix.

If *jobz* = 'N' or 'U', then overwritten by the banded matrix *B* and details of the unitary matrix Q_B to reduce *A* to *B* as specified by *uplo*.

z

On exit,

if *jobz* = 'U', then the matrix *z* is overwritten by *Z*Q* .

If *jobz* = 'N' or 'V', then *z* is not referenced.

d, *e* — REAL for cherdb.

DOUBLE PRECISION for zherdb.

Arrays:

d(*) contains the diagonal elements of the matrix *T*.

The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.

The dimension of *e* must be at least max(1, *n*-1).

tau(*) stores further details of the orthogonal matrix *Q*.

	The dimension of *tau* must be at least max(1, *n-kd*-1).
tau	COMPLEX for cherdb. DOUBLE COMPLEX for zherdb. Array, size at least max(1, *n*-1) Stores further details of the unitary matrix Q_B. The dimension of *tau* must be at least max(1, *n-kd*-1).
work(1)	If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n**(3**kd*+3).

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using *kd* equal to 40 if n ≤ 2000 and 64 otherwise.

Try using ?herdb instead of ?hetrd on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. ?herdb becomes faster beginning approximately with n = 1000, and much faster at larger matrices with a better scalability than ?hetrd.

Avoid applying ?herdb for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to *Z* is doubled compared to the traditional one-step reduction. In that case it is better to apply ?hetrd and ?unmtr/?ungtr to obtain tridiagonal form along with the unitary transformation matrix *Q*.

?orgtr
Generates the real orthogonal matrix Q determined by ?sytrd*.*

Syntax

```
call sorgtr(uplo, n, a, lda, tau, work, lwork, info)
call dorgtr(uplo, n, a, lda, tau, work, lwork, info)
call orgtr(a, tau [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine explicitly generates the *n*-by-*n* orthogonal matrix *Q* formed by `?sytrd` when reducing a real symmetric matrix *A* to tridiagonal form: $A = Q*T*Q^T$. Use this routine after a call to `?sytrd`.

Input Parameters

uplo
: `CHARACTER*1`. Must be `'U'` or `'L'`.

  Use the same *uplo* as supplied to `?sytrd`.

n
: `INTEGER`. The order of the matrix *Q* ($n \geq 0$).

a, *tau*, *work*
: `REAL` for `sorgtr`

  `DOUBLE PRECISION` for `dorgtr`.

  Arrays:

  a(*lda*,*) is the array *a* as returned by `?sytrd`.

  The second dimension of *a* must be at least max(1, *n*).

  tau(*) is the array *tau* as returned by `?sytrd`.

  The size of *tau* must be at least max(1, *n*-1).

  work is a workspace array, its dimension `max(1, lwork)`.

lda
: `INTEGER`. The leading dimension of *a*; at least max(1, *n*).

lwork
: `INTEGER`. The size of the *work* array ($lwork \geq n$).

  If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

  See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a
: Overwritten by the orthogonal matrix *Q*.

work(1)
: If $info = 0$, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info
: `INTEGER`.

  If $info = 0$, the execution is successful.

  If $info = -i$, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `orgtr` interface are the following:

a
: Holds the matrix *A* of size (*n*,*n*).

tau	Holds the vector of length (*n*-1).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* = (*n*–1)***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is ungtr.

?ormtr
Multiplies a real matrix by the real orthogonal matrix Q determined by ?sytrd*.*

Syntax

```
call sormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call dormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call ormtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a real matrix *C* by *Q* or Q^T, where *Q* is the orthogonal matrix *Q* formed by sytrd when reducing a real symmetric matrix *A* to tridiagonal form: *A* = *Q***T***Q*^{T}. Use this routine after a call to ?sytrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products *Q***C*, Q^T**C*, *C***Q*, or *C***Q*^{T} (overwriting the result on *C*).

Input Parameters

In the descriptions below, *r* denotes the order of *Q*:

If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

side	CHARACTER*1. Must be either 'L' or 'R'.

If *side* = 'L', Q or Q^T is applied to C from the left.

If *side* = 'R', Q or Q^T is applied to C from the right.

uplo — CHARACTER*1. Must be 'U' or 'L'.

Use the same *uplo* as supplied to ?sytrd.

trans — CHARACTER*1. Must be either 'N' or 'T'.

If *trans* = 'N', the routine multiplies C by Q.

If *trans* = 'T', the routine multiplies C by Q^T.

m — INTEGER. The number of rows in the matrix C ($m \geq 0$).

n — INTEGER. The number of columns in C ($n \geq 0$).

a, *c*, *tau*, *work* — REAL for sormtr

DOUBLE PRECISION for dormtr

a(*lda*,*) and *tau* are the arrays returned by ?sytrd.

The second dimension of *a* must be at least max(1, *r*).

The size of *tau* must be at least max(1, *r*-1).

c(*ldc*,*) contains the matrix C.

The second dimension of *c* must be at least max(1, *n*)

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; $lda \geq \max(1, r)$.

ldc — INTEGER. The leading dimension of *c*; $ldc \geq \max(1, m)$.

lwork — INTEGER. The size of the *work* array. Constraints:

$lwork \geq \max(1, n)$ if *side* = 'L';

$lwork \geq \max(1, m)$ if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `ormtr` interface are the following:

a	Holds the matrix *A* of size (*r*,*r*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*r*-1).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* for *side* = 'L', or *lwork* = *m*blocksize* for *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $||E||_2 = O(\varepsilon)*||C||_2$.

The total number of floating-point operations is approximately $2*m^2*n$, if *side* = 'L', or $2*n^2*m$, if *side* = 'R'.

The complex counterpart of this routine is unmtr.

?hetrd
Reduces a complex Hermitian matrix to tridiagonal form.

Syntax

```
call chetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call zhetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call hetrd(a, tau [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a complex Hermitian matrix *A* to symmetric tridiagonal form *T* by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix *Q* is not formed explicitly but is represented as a product of *n*-1 elementary reflectors. Routines are provided to work with *Q* in this representation. (They are described later in this topic.)

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

a, *work*

COMPLEX for chetrd

DOUBLE COMPLEX for zhetrd.

a(*lda*,*) is an array containing either upper or lower triangular part of the matrix *A*, as specified by *uplo*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of *a*; at least max(1, *n*).

lwork

INTEGER. The size of the *work* array ($lwork \geq n$).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a

On exit,

if *uplo* = 'U', the diagonal and first superdiagonal of *A* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors;

if *uplo* = 'L', the diagonal and first subdiagonal of *A* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors.

d, *e*	`REAL` for `chetrd` `DOUBLE PRECISION` for `zhetrd`. Arrays: *d*(*) contains the diagonal elements of the matrix *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1).
tau	`COMPLEX` for `chetrd``DOUBLE COMPLEX` for `zhetrd`. Array, size at least max(1, *n*-1). Stores (*n*-1) scalars that define elementary reflectors in decomposition of the unitary matrix *Q* in a product of *n*-1 elementary reflectors.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hetrd` interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*-1).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Note that diagonal (*d*) and off-diagonal (*e*) elements of the matrix *T* are omitted because they are kept in the matrix *A* on exit.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where $||E||_2 = c(n)*\varepsilon*||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

ungtr	to form the computed matrix *Q* explicitly
unmtr	to multiply a complex matrix by *Q*.

The real counterpart of this routine is ?sytrd.

?ungtr
Generates the complex unitary matrix Q determined by ?hetrd*.*

Syntax

```
call cungtr(uplo, n, a, lda, tau, work, lwork, info)
call zungtr(uplo, n, a, lda, tau, work, lwork, info)
call ungtr(a, tau [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine explicitly generates the *n*-by-*n* unitary matrix *Q* formed by ?hetrd when reducing a complex Hermitian matrix *A* to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to ?hetrd.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?hetrd.
n	INTEGER. The order of the matrix *Q* ($n \geq 0$).
a, *tau*, *work*	COMPLEX for cungtr DOUBLE COMPLEX for zungtr. Arrays: *a*(*lda*,*) is the array *a* as returned by ?hetrd. The second dimension of *a* must be at least max(1, *n*). *tau*(*) is the array *tau* as returned by ?hetrd. The dimension of *tau* must be at least max(1, *n*-1). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array ($lwork \geq n$).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the unitary matrix *Q*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ungtr interface are the following:

a — Holds the matrix *A* of size (*n*,*n*).

tau — Holds the vector of length (*n*-1).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* = (*n*-1)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is orgtr.

?unmtr
Multiplies a complex matrix by the complex unitary matrix Q determined by `?hetrd`.

Syntax

```
call cunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call zunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call unmtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine multiplies a complex matrix C by Q or Q^H, where Q is the unitary matrix Q formed by `?hetrd` when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to `?hetrd`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q:

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

side
`CHARACTER*1`. Must be either 'L' or 'R'.
If *side* = 'L', Q or Q^H is applied to C from the left.
If *side* = 'R', Q or Q^H is applied to C from the right.

uplo
`CHARACTER*1`. Must be 'U' or 'L'.
Use the same *uplo* as supplied to `?hetrd`.

trans
`CHARACTER*1`. Must be either 'N' or 'T'.
If *trans* = 'N', the routine multiplies C by Q.
If *trans* = 'C', the routine multiplies C by Q^H.

m
`INTEGER`. The number of rows in the matrix C ($m \geq 0$).

n
`INTEGER`. The number of columns in C ($n \geq 0$).

a, *c*, *tau*, *work*
`COMPLEX` for `cunmtr`
`DOUBLE COMPLEX` for `zunmtr`.
a(*lda*,*) and *tau* are the arrays returned by `?hetrd`.
The second dimension of *a* must be at least max(1, r).
The dimension of *tau* must be at least max(1, r-1).
c(*ldc*,*) contains the matrix C.
The second dimension of *c* must be at least max(1, n)

work is a workspace array, its dimension `max(1, lwork)`.

`lda` — `INTEGER`. The leading dimension of *a*; `lda` ≥ `max(1, r)`.

`ldc` — `INTEGER`. The leading dimension of *c*; `ldc` ≥ `max(1, n)` .

`lwork` — `INTEGER`. The size of the *work* array. Constraints:

`lwork` ≥ `max(1, n)` if `side = 'L'`;

`lwork` ≥ `max(1, m)` if `side = 'R'`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

`c` — Overwritten by the product $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (as specified by *side* and `trans`).

`work(1)` — If `info = 0`, on exit `work(1)` contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

`info` — `INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `unmtr` interface are the following:

`a` — Holds the matrix *A* of size (*r*,*r*).

`r = m` if `side = 'L'`.

`r = n` if `side = 'R'`.

`tau` — Holds the vector of length (*r*-1).

`c` — Holds the matrix *C* of size (*m*,*n*).

`side` — Must be `'L'` or `'R'`. The default value is `'L'`.

`uplo` — Must be `'U'` or `'L'`. The default value is `'U'`.

`trans` — Must be `'N'` or `'C'`. The default value is `'N'`.

Application Notes

For better performance, try using `lwork = n*blocksize` (for `side = 'L'`) or `lwork = m*blocksize` (for `side = 'R'`) where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $||E||_2 = O(\varepsilon)*||C||_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if `side = 'L'` or $8*n^2*m$ if `side = 'R'`.

The real counterpart of this routine is ormtr.

?orm22/?unm22
Multiplies a general matrix by an orthogonal/unitary matrix with a 2x2 structure.

Syntax

```
call sorm22 (side, trans, m, n, n1, n2, q, ldq, c, ldc, work, lwork, info )
call dorm22 (side, trans, m, n, n1, n2, q, ldq, c, ldc, work, lwork, info )
call cunm22(side, trans, m, n, n1, n2, q, ldq, c, ldc, work, lwork, info)
call zunm22(side, trans, m, n, n1, n2, q, ldq, c, ldc, work, lwork, info)
```

Include Files

- `mkl.fi`

Description

`?orm22/?unm22` overwrites the general real/complex *m*-by-*n* matrix *C* with

	`side` = 'L'	`side` = 'R'
`trans` = 'N'	*Q* * *C*	*C* * *Q*
`trans` = 'T' applies to `sorm22` and `dorm22` only	Q^T * *C*	*C* * Q^T
`trans` = 'C' applies to `cunm22` and `zunm22` only	Q^H * *C*	*C* * Q^H

where *Q* is a real orthogonal/complex unitary matrix of order *nq*, with *nq* = *m* if `side` = 'L' and *nq* = *n* if `side` = 'R'.

The orthogonal/unitary matrix *Q* processes a 2-by-2 block structure:

$$Q = \begin{pmatrix} Q11 & Q12 \\ Q21 & Q22 \end{pmatrix}$$

where *Q12* is an *n1*-by-*n1* lower triangular matrix and *Q21* is an *n2*-by-*n2* upper triangular matrix.

Input Parameters

side
CHARACTER*1. = 'L': apply *Q*, Q^T, or Q^H from the left;
= 'R': apply *Q*, Q^T, or Q^H from the right.

trans
CHARACTER*1. = 'N': apply *Q* (no transpose);
= 'T': apply Q^T (transpose) - `sorm22` and `dorm22` only;
= 'C': apply Q^H (conjugate transpose) - `cunm22` and `zunm22` only.

m
INTEGER. The number of rows of the matrix *C*.
m≥ 0.

n
INTEGER. The number of columns of the matrix *C*.
n≥ 0.

n1
INTEGER. The dimension of *Q12*.
n1≥ 0.
The following requirement must be satisfied: *n1* + *n2* = *m* if *side* = 'L' and *n1* + *n2* = *n* if *side* = 'R'.

n2
INTEGER. The dimension of *Q21*.
n2≥ 0.
The following requirement must be satisfied: *n1* + *n2* = *m* if *side* = 'L' and *n1* + *n2* = *n* if *side* = 'R'.

q
`REAL` for `sorm22`
`DOUBLE PRECISION` for `dorm22`
`COMPLEX` for `cunm22`
`DOUBLE COMPLEX` for `zunm22`
Array, size (*ldq*,*m*) if *side* = 'L' and (*ldq*,*n*) if *side* = 'R'.

ldq
INTEGER. The leading dimension of the array *q*.
ldq≥ max(1,*m*) if *side* = 'L';
ldq≥ max(1,*n*) if *side* = 'R'.

c
`REAL` for `sorm22`
`DOUBLE PRECISION` for `dorm22`
`COMPLEX` for `cunm22`
`DOUBLE COMPLEX` for `zunm22`
Array, size (*ldc*,*n*)
On entry, the *m*-by-*n* matrix *C*.

ldc
INTEGER. The leading dimension of the array *c*.
ldc≥ max(1,*m*).

lwork INTEGER. The dimension of the array *work*.

If *side* = 'L', *lwork*≥ max(1,*n*);

if *side* = 'R', *lwork*≥ max(1,*m*).

For optimum performance *lwork*≥*m**n.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

c On exit, *c* is overwritten by the product:

*Q***C*,

Q^T**C*,

Q^H * *C*,

*C**Q^T,

C * Q^H, or

*C**Q.

work REAL for sorm22

DOUBLE PRECISION for dorm22

COMPLEX for cunm22

DOUBLE COMPLEX for zunm22

Array, size (max(1,*lwork*))

On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info INTEGER. = 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?sptrd
Reduces a real symmetric matrix to tridiagonal form using packed storage.

Syntax

```
call ssptrd(uplo, n, ap, d, e, tau, info)
call dsptrd(uplo, n, ap, d, e, tau, info)
call sptrd(ap, tau [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a packed real symmetric matrix *A* to symmetric tridiagonal form *T* by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix *Q* is not formed explicitly but is represented as a product of *n*-1 elementary reflectors. Routines are provided for working with *Q* in this representation. See *Application Notes* below for details.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangle of *A*.

If *uplo* = 'L', *ap* stores the packed lower triangle of *A*.

n — INTEGER. The order of the matrix *A* ($n \geq 0$).

ap — REAL for ssptrd

DOUBLE PRECISION for dsptrd.

Array, size at least max(1, *n*(*n*+1)/2). Contains either upper or lower triangle of *A* (as specified by *uplo*) in the packed form described in Matrix Storage Schemes.

Output Parameters

ap — Overwritten by the tridiagonal matrix *T* and details of the orthogonal matrix *Q*, as specified by *uplo*.

d, *e*, *tau* — REAL for ssptrd

DOUBLE PRECISION for dsptrd.

Arrays:

d(*) contains the diagonal elements of the matrix *T*.

The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.

The dimension of *e* must be at least max(1, *n*-1).

tau(*) Stores (*n*-1) scalars that define elementary reflectors in decomposition of the matrix *Q* in a product of *n*-1 reflectors.

The dimension of *tau* must be at least max(1, *n*-1).

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sptrd interface are the following:

ap — Holds the array *A* of size (*n**(*n*+1)/2).

tau — Holds the vector with the number of elements *n*-1.

uplo	Must be 'U' or 'L'. The default value is 'U'.

Note that diagonal (*d*) and off-diagonal (*e*) elements of the matrix *T* are omitted because they are kept in the matrix *A* on exit.

Application Notes

The matrix *Q* is represented as a product of *n*-1 *elementary reflectors*, as follows :

- If *uplo* = 'U', $Q = H(n-1) \ldots H(2)H(1)$

  Each *H*(i) has the form

  $H(i) = I - tau*v*v^T$

  where *tau* is a real scalar and *v* is a real vector with $v(i+1:n) = 0$ and $v(i) = 1$.

  On exit, *tau* is stored in *tau*(i), and $v(1:i-1)$ is stored in *AP*, overwriting $A(1:i-1, i+1)$.
- If *uplo* = 'L', $Q = H(1)H(2) \ldots H(n-1)$

  Each *H*(i) has the form

  $H(i) = I - tau*v*v^T$

  where *tau* is a real scalar and *v* is a real vector with $v(1:i) = 0$ and $v(i+1) = 1$.

  On exit, *tau* is stored in *tau*(i), and $v(i+2:n)$ is stored in *AP*, overwriting $A(i+2:n, i)$.

The computed matrix *T* is exactly similar to a matrix $A+E$, where $||E||_2 = c(n)*\varepsilon*||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision. The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

opgtr	to form the computed matrix *Q* explicitly
opmtr	to multiply a real matrix by *Q*.

The complex counterpart of this routine is hptrd.

?opgtr
Generates the real orthogonal matrix Q determined by ?sptrd*.*

Syntax

```
call sopgtr(uplo, n, ap, tau, q, ldq, work, info)
call dopgtr(uplo, n, ap, tau, q, ldq, work, info)
call opgtr(ap, tau, q [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine explicitly generates the *n*-by-*n* orthogonal matrix *Q* formed by sptrd when reducing a packed real symmetric matrix *A* to tridiagonal form: $A = Q*T*Q^T$. Use this routine after a call to ?sptrd.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?sptrd.
n	INTEGER. The order of the matrix *Q* ($n \geq 0$).
ap, *tau*	REAL for sopgtr DOUBLE PRECISION for dopgtr. Arrays *ap* and *tau*, as returned by ?sptrd. The size of *ap* must be at least max(1, $n(n+1)/2$). The size of *tau* must be at least max(1, $n-1$).
ldq	INTEGER. The leading dimension of the output array *q*; at least max(1, *n*).
work	REAL for sopgtr DOUBLE PRECISION for dopgtr. Workspace array, size at least max(1, $n-1$).

Output Parameters

q	REAL for sopgtr DOUBLE PRECISION for dopgtr. Array, size (*ldq*,*) . Contains the computed matrix *Q*. The second dimension of *q* must be at least max(1, *n*).
info	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine opgtr interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
tau	Holds the vector with the number of elements *n* - 1.
q	Holds the matrix *Q* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is upgtr.

?opmtr

Multiplies a real matrix by the real orthogonal matrix Q determined by ?sptrd.

Syntax

```
call sopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call dopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call opmtr(ap, tau, c [,side] [,uplo] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q formed by sptrd when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q*T*Q^T$. Use this routine after a call to ?sptrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q:

If side = 'L', $r = m$; if side = 'R', $r = n$.

side
CHARACTER*1. Must be either 'L' or 'R'.
If side = 'L', Q or Q^T is applied to C from the left.
If side = 'R', Q or Q^T is applied to C from the right.

uplo
CHARACTER*1. Must be 'U' or 'L'.
Use the same *uplo* as supplied to ?sptrd.

trans
CHARACTER*1. Must be either 'N' or 'T'.
If trans = 'N', the routine multiplies C by Q.
If trans = 'T', the routine multiplies C by Q^T.

m
INTEGER. The number of rows in the matrix C ($m \geq 0$).

n
INTEGER. The number of columns in C ($n \geq 0$).

ap, *tau*, *c*, *work*
REAL for sopmtr
DOUBLE PRECISION for dopmtr.
ap and *tau* are the arrays returned by ?sptrd.
The dimension of *ap* must be at least max(1, $r(r+1)/2$).
The dimension of *tau* must be at least max(1, r-1).
c(*ldc*,*) contains the matrix C.

The second dimension of *c* must be at least max(1, *n*)

work(*) is a workspace array.

The dimension of *work* must be at least

max(1, *n*) if *side* = 'L';

max(1, *m*) if *side* = 'R'.

ldc INTEGER. The leading dimension of *c*; *ldc*≥ max(1, *n*) .

Output Parameters

c Overwritten by the product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine opmtr interface are the following:

ap Holds the array *A* of size (*r**(r+1)/2), where

r = *m* if *side* = 'L'.

r = *n* if *side* = 'R'.

tau Holds the vector with the number of elements *r* - 1.

c Holds the matrix *C* of size (*m*,*n*).

side Must be 'L' or 'R'. The default value is 'L'.

uplo Must be 'U' or 'L'. The default value is 'U'.

trans Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

The computed product differs from the exact product by a matrix *E* such that $||E||_2 = O(\varepsilon) ||C||_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $2*m^2*n$ if *side* = 'L', or $2*n^2*m$ if *side* = 'R'.

The complex counterpart of this routine is upmtr.

?hptrd

Reduces a complex Hermitian matrix to tridiagonal form using packed storage.

Syntax

```
call chptrd(uplo, n, ap, d, e, tau, info)
call zhptrd(uplo, n, ap, d, e, tau, info)
call hptrd(ap, tau [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a packed complex Hermitian matrix *A* to symmetric tridiagonal form *T* by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix *Q* is not formed explicitly but is represented as a product of *n*-1 elementary reflectors. Routines are provided for working with *Q* in this representation (see *Application Notes* below).

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangle of *A*.

If *uplo* = 'L', *ap* stores the packed lower triangle of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

ap

COMPLEX for chptrd

DOUBLE COMPLEX for zhptrd.

Array, size at least max(1, *n*(*n*+1)/2). Contains either upper or lower triangle of *A* (as specified by *uplo*) in the packed form described in "Matrix Storage Schemes.

Output Parameters

ap

Overwritten by the tridiagonal matrix *T* and details of the unitary matrix *Q*, as specified by *uplo*.

d, *e*

REAL for chptrd

DOUBLE PRECISION for zhptrd.

Arrays:

d(*) contains the diagonal elements of the matrix *T*.

The size of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.

The size of *e* must be at least max(1, *n*-1).

tau

COMPLEX for chptrd

DOUBLE COMPLEX for zhptrd.

Array, size at least max(1, *n*-1). Stores (n-1) scalars that define elementary reflectors in decomposition of the unitary matrix *Q* in a product of reflectors.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hptrd interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
tau	Holds the vector with the number of elements *n* - 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Note that diagonal (*d*) and off-diagonal (*e*) elements of the matrix *T* are omitted because they are kept in the matrix *A* on exit.

Application Notes

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where $||E||_2 = c(n)*\varepsilon*||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

upgtr	to form the computed matrix *Q* explicitly
upmtr	to multiply a complex matrix by *Q*.

The real counterpart of this routine is sptrd.

?upgtr
Generates the complex unitary matrix Q determined by ?hptrd*.*

Syntax

```
call cupgtr(uplo, n, ap, tau, q, ldq, work, info)
call zupgtr(uplo, n, ap, tau, q, ldq, work, info)
call upgtr(ap, tau, q [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine explicitly generates the *n*-by-*n* unitary matrix *Q* formed by hptrd when reducing a packed complex Hermitian matrix *A* to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to ?hptrd.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?hptrd.
n	INTEGER. The order of the matrix *Q* ($n \geq 0$).
ap, *tau*	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Arrays *ap* and *tau*, as returned by ?hptrd. The dimension of *ap* must be at least max(1, $n(n+1)/2$). The dimension of *tau* must be at least max(1, $n-1$).
ldq	INTEGER. The leading dimension of the output array *q*; at least max(1, *n*).
work	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Workspace array, size at least max(1, $n-1$).

Output Parameters

q	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Array, size (*ldq*,*) . Contains the computed matrix *Q*. The second dimension of *q* must be at least max(1, *n*).
info	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine upgtr interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
tau	Holds the vector with the number of elements *n* - 1.
q	Holds the matrix *Q* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is `(16/3)n3`.

The real counterpart of this routine is opgtr.

?upmtr

Multiplies a complex matrix by the unitary matrix Q determined by `?hptrd`.

Syntax

```
call cupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call zupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call upmtr(ap, tau, c [,side] [,uplo] [,trans] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine multiplies a complex matrix C by Q or Q^H, where Q is the unitary matrix formed by hptrd when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to `?hptrd`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q:

If `side = 'L'`, $r = m$; if `side = 'R'`, $r = n$.

side	`CHARACTER*1`. Must be either `'L'` or `'R'`. If `side = 'L'`, Q or Q^H is applied to C from the left. If `side = 'R'`, Q or Q^H is applied to C from the right.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Use the same *uplo* as supplied to `?hptrd`.
trans	`CHARACTER*1`. Must be either `'N'` or `'T'`. If `trans = 'N'`, the routine multiplies C by Q. If `trans = 'T'`, the routine multiplies C by Q^H.
m	`INTEGER`. The number of rows in the matrix C ($m \geq 0$).
n	`INTEGER`. The number of columns in C ($n \geq 0$).
ap, *tau*, *c*,	`COMPLEX` for `cupmtr` `DOUBLE COMPLEX` for `zupmtr`. *ap* and *tau* are the arrays returned by `?hptrd`. The size of *ap* must be at least max(1, $r(r+1)/2$). The size of *tau* must be at least max(1, $r-1$).

c(*ldc*,*) contains the matrix *C*.

The second dimension of *c* must be at least max(1, *n*)

work(*) is a workspace array.

The dimension of *work* must be at least

max(1, *n*) if `side = 'L'`;

max(1, *m*) if `side = 'R'`.

`ldc` INTEGER. The leading dimension of *c*; `ldc` ≥ `max(1, m)` .

Output Parameters

`c` Overwritten by the product *Q***C*, Q^H**C*, *C***Q*, or *C**Q^H (as specified by *side* and *trans*).

`info` INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `upmtr` interface are the following:

`ap` Holds the array *A* of size (*r**(*r*+1)/2), where

`r = m` if `side = 'L'`.

`r = n` if `side = 'R'`.

`tau` Holds the vector with the number of elements *n* - 1.

`c` Holds the matrix *C* of size (*m*,*n*).

`side` Must be `'L'` or `'R'`. The default value is `'L'`.

`uplo` Must be `'U'` or `'L'`.The default value is `'U'`.

`trans` Must be `'N'` or `'C'`. The default value is `'N'`.

Application Notes

The computed product differs from the exact product by a matrix *E* such that $||E||_2 = O(\varepsilon)*||C||_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if `side = 'L'` or $8*n^2*m$ if `side = 'R'`.

The real counterpart of this routine is opmtr.

?sbtrd
Reduces a real symmetric band matrix to tridiagonal form.

Syntax

```
call ssbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call dsbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call sbtrd(ab[, q] [,vect] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a real symmetric band matrix *A* to symmetric tridiagonal form *T* by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix *Q* is determined as a product of Givens rotations.

If required, the routine can also form the matrix *Q* explicitly.

Input Parameters

vect

CHARACTER*1. Must be 'V', 'N', or 'U'.

If *vect* = 'V', the routine returns the explicit matrix *Q*.

If *vect* = 'N', the routine does not return *Q*.

If *vect* = 'U', the routine updates matrix *X* by forming *X***Q*.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ab* stores the upper triangular part of *A*.

If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

kd

INTEGER. The number of super- or sub-diagonals in *A*

($kd \geq 0$).

ab, *q*, *work*

REAL for ssbtrd

DOUBLE PRECISION for dsbtrd.

ab(*ldab*,*) is an array containing either upper or lower triangular part of the matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of *ab* must be at least max(1, *n*).

q(*ldq*,*) is an array.

If *vect* = 'U', the *q* array must contain an *n*-by-*n* matrix *X*.

If *vect* = 'N' or 'V', the *q* parameter need not be set.

The second dimension of *q* must be at least max(1, *n*).

work(*) is a workspace array.

The dimension of *work* must be at least max(1, *n*).

ldab

INTEGER. The leading dimension of *ab*; at least *kd*+1 .

ldq INTEGER. The leading dimension of *q*. Constraints:

ldq ≥ max(1, *n*) if *vect* = 'V' or 'U';

ldq ≥ 1 if *vect* = 'N'.

Output Parameters

ab On exit, the diagonal elements of the array *ab* are overwritten by the diagonal elements of the tridiagonal matrix *T*. If *kd* > 0, the elements on the first superdiagonal (if *uplo* = 'U') or the first subdiagonal (if *uplo* = 'L') are ovewritten by the off-diagonal elements of *T*. The rest of *ab* is overwritten by values generated during the reduction.

d, *e*, *q* REAL for ssbtrd

DOUBLE PRECISION for dsbtrd.

Arrays:

d(*) contains the diagonal elements of the matrix *T*.

The size of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.

The size of *e* must be at least max(1, *n*-1).

q(*ldq*,*) is not referenced if *vect* = 'N'.

If *vect* = 'V', *q* contains the *n*-by-*n* matrix *Q*.

If *vect* = 'U', *q* contains the product *X** *Q*.

The second dimension of *q* must be:

at least max(1, *n*) if *vect* = 'V';

at least 1 if *vect* = 'N'.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sbtrd interface are the following:

ab Holds the array *A* of size (*kd*+1,*n*).

q Holds the matrix *Q* of size (*n*,*n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

vect If omitted, this argument is restored based on the presence of argument *q* as follows: *vect* = 'V', if *q* is present, *vect* = 'N', if *q* is omitted.

If present, *vect* must be equal to 'V' or 'U' and the argument *q* must also be present. Note that there will be an error condition if *vect* is present and *q* omitted.

Note that diagonal (*d*) and off-diagonal (*e*) elements of the matrix *T* are omitted because they are kept in the matrix *A* on exit.

Application Notes

The computed matrix *T* is exactly similar to a matrix *A*+*E*, where $||E||_2 = c(n)*\varepsilon*||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision. The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$.

The total number of floating-point operations is approximately $6n^2*kd$ if *vect* = 'N', with $3n^3*(kd-1)/kd$ additional operations if *vect* = 'V'.

The complex counterpart of this routine is hbtrd.

?hbtrd
Reduces a complex Hermitian band matrix to tridiagonal form.

Syntax

```
call chbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call zhbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call hbtrd(ab [, q] [,vect] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a complex Hermitian band matrix *A* to symmetric tridiagonal form *T* by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix *Q* is determined as a product of Givens rotations.

If required, the routine can also form the matrix *Q* explicitly.

Input Parameters

vect — CHARACTER*1. Must be 'V', 'N', or 'U'.
If *vect* = 'V', the routine returns the explicit matrix *Q*.
If *vect* = 'N', the routine does not return *Q*.
If *vect* = 'U', the routine updates matrix *X* by forming *Q***X*.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *ab* stores the upper triangular part of *A*.
If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n — INTEGER. The order of the matrix *A* ($n \geq 0$).

kd — INTEGER. The number of super- or sub-diagonals in *A*

($kd \geq 0$).

ab, *work*

COMPLEX for chbtrd

DOUBLE COMPLEX for zhbtrd.

ab(*ldab*,*) is an array containing either upper or lower triangular part of the matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of *ab* must be at least max(1, *n*).

work(*) is a workspace array.

The dimension of *work* must be at least max(1, *n*).

q

COMPLEX for chbtrd

DOUBLE COMPLEX for zhbtrd.

q(*ldq*,*) is an array.

If *vect* = 'U', the *q* array must contain an *n*-by-*n* matrix *X*.

If *vect* = 'N' or 'V', the *q* parameter need not be set.'

ldab

INTEGER. The leading dimension of *ab*; at least *kd*+1.

ldq

INTEGER. The leading dimension of *q*. Constraints:

ldq ≥ max(1, *n*) if *vect* = 'V' or 'U';

ldq ≥ 1 if *vect* = 'N'.

Output Parameters

ab

On exit, the diagonal elements of the array *ab* are overwritten by the diagonal elements of the tridiagonal matrix *T*. If *kd* > 0, the elements on the first superdiagonal (if *uplo* = 'U') or the first subdiagonal (if *uplo* = 'L') are ovewritten by the off-diagonal elements of *T*. The rest of *ab* is overwritten by values generated during the reduction.

d, *e*

REAL for chbtrd

DOUBLE PRECISION for zhbtrd.

Arrays:

d(*) contains the diagonal elements of the matrix *T*.

The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.

The dimension of *e* must be at least max(1, *n*-1).

q

If *vect* = 'N', *q* is not referenced.

If *vect* = 'V', *q* contains the *n*-by-*n* matrix *Q*.

If *vect* = 'U', *q* contains the product *X** *Q*.

The second dimension of *q* must be:

at least max(1, *n*) if *vect* = 'V';

at least 1 if *vect* = 'N'.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hbtrd interface are the following:

ab	Holds the array *A* of size (*kd*+1,*n*).
q	Holds the matrix *Q* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
vect	If omitted, this argument is restored based on the presence of argument *q* as follows: *vect* = 'V', if *q* is present, *vect* = 'N', if *q* is omitted. If present, *vect* must be equal to 'V' or 'U' and the argument *q* must also be present. Note that there will be an error condition if *vect* is present and *q* omitted.

Note that diagonal (*d*) and off-diagonal (*e*) elements of the matrix *T* are omitted because they are kept in the matrix *A* on exit.

Application Notes

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where $||E||_2 = c(n)*\varepsilon*||A||_2$, *c*(*n*) is a modestly increasing function of *n*, and ε is the machine precision. The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$.

The total number of floating-point operations is approximately $20n^2*kd$ if *vect* = 'N', with $10n^3*(kd-1)/kd$ additional operations if *vect* = 'V'.

The real counterpart of this routine is sbtrd.

?sterf

Computes all eigenvalues of a real symmetric tridiagonal matrix using QR algorithm.

Syntax

```
call ssterf(n, d, e, info)
call dsterf(n, d, e, info)
call sterf(d, e [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues of a real symmetric tridiagonal matrix *T* (which can be obtained by reducing a symmetric or Hermitian matrix to tridiagonal form). The routine uses a square-root-free variant of the *QR* algorithm.

If you need not only the eigenvalues but also the eigenvectors, call steqr.

Input Parameters

n	INTEGER. The order of the matrix *T* ($n \geq 0$).
d, *e*	REAL for ssterf DOUBLE PRECISION for dsterf. Arrays: *d*(*) contains the diagonal elements of *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1).

Output Parameters

d	The *n* eigenvalues in ascending order, unless *info* > 0. See also *info*.
e	On exit, the array is overwritten; see *info*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, the algorithm failed to find all the eigenvalues after 30*n* iterations: *i* off-diagonal elements have not converged to zero. On exit, *d* and *e* contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to *T*. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran 95 Interface Conventions.

Specific details for the routine sterf interface are the following:

d	Holds the vector of length *n*.
e	Holds the vector of length (*n*-1).

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $||E||_2 = O(\varepsilon)*||T||_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and m_i is the corresponding computed value, then

$|\mu i - \lambda i| \leq c(n)*\varepsilon*||T||_2$

where $c(n)$ is a modestly increasing function of n.

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about $14n^2$.

?steqr

Computes all eigenvalues and eigenvectors of a symmetric or Hermitian matrix reduced to tridiagonal form (QR algorithm).

Syntax

```
call ssteqr(compz, n, d, e, z, ldz, work, info)
call dsteqr(compz, n, d, e, z, ldz, work, info)
call csteqr(compz, n, d, e, z, ldz, work, info)
call zsteqr(compz, n, d, e, z, ldz, work, info)
call rsteqr(d, e [,z] [,compz] [,info])
call steqr(d, e [,z] [,compz] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric tridiagonal matrix T. In other words, the routine can compute the spectral factorization: $T = Z*\Lambda*Z^T$. Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$T*z_i = \lambda_i*z_i$ for $i = 1, 2, \ldots, n$.

The routine normalizes the eigenvectors so that $||z_i||_2 = 1$.

You can also use the routine for computing the eigenvalues and eigenvectors of an arbitrary real symmetric (or complex Hermitian) matrix A reduced to tridiagonal form T: $A = Q*T*Q^H$. In this case, the spectral factorization is as follows: $A = Q*T*Q^H = (Q*Z)*\Lambda*(Q*Z)^H$. Before calling ?steqr, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	?sytrd, ?orgtr	?hetrd, ?ungtr
packed storage	?sptrd, ?opgtr	?hptrd, ?upgtr
band storage	?sbtrd(*vect*='V')	?hbtrd(*vect*='V')

If you need eigenvalues only, it's more efficient to call sterf. If T is positive-definite, pteqr can compute small eigenvalues more accurately than ?steqr.

To solve the problem by a single call, use one of the divide and conquer routines stevd, syevd, spevd, or sbevd for real symmetric matrices or heevd, hpevd, or hbevd for complex Hermitian matrices.

Input Parameters

compz

CHARACTER*1. Must be 'N' or 'I' or 'V'.

If *compz* = 'N', the routine computes eigenvalues only.

If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix *T*.

If *compz* = 'V', the routine computes the eigenvalues and eigenvectors of the original symmetric matrix. On entry, *z* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.

n

INTEGER. The order of the matrix *T* (*n*≥ 0).

d, *e*, *work*

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays:

d(*) contains the diagonal elements of *T*.

The size of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.

The size of *e* must be at least max(1, *n*-1).

work(*) is a workspace array.

The size of *work* must be:

at least 1 if *compz* = 'N';

at least max(1, 2**n*-2) if *compz* = 'V' or 'I'.

z

REAL for ssteqr

DOUBLE PRECISION for dsteqr

COMPLEX for csteqr

DOUBLE COMPLEX for zsteqr.

Array, size (*ldz*, *).

If *compz* = 'N' or 'I', *z* need not be set.

If *vect* = 'V', *z* must contain the orthogonal matrix used in the reduction to tridiagonal form.

The second dimension of *z* must be:

at least 1 if *compz* = 'N';

at least max(1, *n*) if *compz* = 'V' or 'I'.

work (*lwork*) is a workspace array.

ldz

INTEGER. The leading dimension of *z*. Constraints:

ldz≥ 1 if *compz* = 'N';

ldz≥ max(1, *n*) if *compz* = 'V' or 'I'.

Output Parameters

d	The *n* eigenvalues in ascending order, unless *info* > 0. See also *info*.
e	On exit, the array is overwritten; see *info*.
z	If *info* = 0, contains the *n*-by-*n* matrix the columns of which are orthonormal eigenvectors (the *i*-th column corresponds to the *i*-th eigenvalue).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, the algorithm failed to find all the eigenvalues after 30*n* iterations: *i* off-diagonal elements have not converged to zero. On exit, *d* and *e* contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to *T*. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine steqr interface are the following:

d	Holds the vector of length *n*.
e	Holds the vector of length (*n*-1).
z	Holds the matrix *Z* of size (*n*,*n*).
compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = 'I', if *z* is present, *compz* = 'N', if *z* is omitted. If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted. Note that two variants of Fortran 95 interface for steqr routine are needed because of an ambiguous choice between real and complex cases appear when *z* is omitted. Thus, the name rsteqr is used in real cases (single or double precision), and the name steqr is used in complex cases (single or double precision).

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $||E||_2 = O(\varepsilon)*||T||_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$|\mu_i - \lambda_i| \leq c(n)*\varepsilon*||T||_2$

where $c(n)$ is a modestly increasing function of n.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$\theta(z_i, w_i) \leq c(n)*\varepsilon*||T||_2 / \min_{i \neq j}|\lambda_i - \lambda_j|$.

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$24n^2$ if *compz* = 'N';

$7n^3$ (for complex flavors, $14n^3$) if *compz* = 'V' or 'I'.

?stemr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)
```

```
call dstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)
```

```
call cstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)
```

```
call zstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*. Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval (vl,vu] or a range of indices il:iu for the desired eigenvalues.

Depending on the number of desired eigenvalues, these are computed either by bisection or the *dqds* algorithm. Numerically orthogonal eigenvectors are computed by the use of various suitable $L*D*L^T$ factorizations near clusters of close eigenvalues (referred to as RRRs, Relatively Robust Representations). An informal sketch of the algorithm follows.

For each unreduced block (submatrix) of *T*,

a. Compute $T - sigma*I = L*D*L^T$, so that *L* and *D* define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of *L* and *D* cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix *T* does not have this property in general.

b. Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see steps c and d.

c. For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.

d. For each eigenvalue with a large enough relative separation compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to step c for any clusters that remain.

Normal execution of `?stemr` may create NaNs and infinities and may abort due to a floating point exception in environments that do not handle NaNs and infinities in the IEEE standard default manner.

For more details, see: [Dhillon04], [Dhillon04-02], [Dhillon97]

Input Parameters

jobz

`CHARACTER*1`. Must be `'N'` or `'V'`.

If *jobz* = `'N'`, then only eigenvalues are computed.

If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

range

`CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.

If *range* = `'A'`, the routine computes all eigenvalues.

If *range* = `'V'`, the routine computes all eigenvalues in the half-open interval: `(vl, vu]`.

If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

n

`INTEGER`. The order of the matrix *T* (*n*≥0).

d

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size (*n*).

Contains *n* diagonal elements of the tridiagonal matrix *T*.

e

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, size *n*.

Contains `(n-1)` off-diagonal elements of the tridiagonal matrix *T* in elements `1` to *n*–`1` of *e*. *e*`(n)` need not be set on input, but is used internally as workspace.

vl, *vu*

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

If *range* = `'V'`, the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*<*vu*.

If *range* = `'A'` or `'I'`, *vl* and *vu* are not referenced.

il, *iu*

`INTEGER`.

If *range* = `'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: `1`≤*il*≤*iu*≤*n*, if *n*>`0`.

If *range* = `'A'` or `'V'`, *il* and *iu* are not referenced.

ldz

`INTEGER`. The leading dimension of the output array *z*.

if *jobz* = `'V'`, then *ldz* ≥ `max(1, n)` ;

ldz ≥ 1 otherwise.

nzc INTEGER. The number of eigenvectors to be held in the array *z*.

If *range* = 'A', then *nzc*≥max(1, *n*);

If *range* = 'V', then *nzc* is greater than or equal to the number of eigenvalues in the half-open interval: (*vl*, *vu*].

If *range* = 'I', then *nzc*≥*iu*-*il*+1.

If *nzc* = -1, then a workspace query is assumed; the routine calculates the number of columns of the array *z* that are needed to hold the eigenvectors.

This value is returned as the first entry of the array *z*, and no error message related to *nzc* is issued by the routine xerbla.

tryrac LOGICAL.

If *tryrac*= .TRUE. is true, it indicates that the code should check whether the tridiagonal matrix defines its eigenvalues to high relative accuracy. If so, the code uses relative-accuracy preserving algorithms that might be (a bit) slower depending on the matrix. If the matrix does not define its eigenvalues to high relative accuracy, the code can uses possibly faster algorithms.

If *tryrac*= .FALSE. is not true, the code is not required to guarantee relatively accurate eigenvalues and can use the fastest possible techniques.

work REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size (*lwork*).

lwork INTEGER.

The dimension of the array *work*,

lwork ≥ max(1, 18*n).

If *lwork*=-1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

iwork INTEGER.

Workspace array, size (*liwork*).

liwork INTEGER.

The dimension of the array *iwork*.

lwork≥max(1, 10*n) if the eigenvectors are desired, and *lwork*≥max(1, 8*n) if only the eigenvalues are to be computed.

If *liwork*=-1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla.

Output Parameters

d

On exit, the array *d* is overwritten.

e

On exit, the array *e* is overwritten.

m

INTEGER.

The total number of eigenvalues found, $0 \le m \le n$.

If *range* = 'A', then $m=n$, and if *range* = 'I', then $m=iu-il+1$.

w

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*).

The first *m* elements contain the selected eigenvalues in ascending order.

z

REAL for sstemr

DOUBLE PRECISION for dstemr

COMPLEX for cstemr

DOUBLE COMPLEX for zstemr.

Array *z*(*ldz*,*), the second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', and *info* = 0, then the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the i-th column of *z* holding the eigenvector associated with *w*(i).

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* = 'V', the exact value of *m* is not known in advance and can be computed with a workspace query by setting *nzc*=-1, see description of the parameter *nzc*.

isuppz

INTEGER.

Array, size (2*max(1, *m*)).

The support of the eigenvectors in *z*, that is the indices indicating the nonzero elements in *z*. The i-th computed eigenvector is nonzero only in elements *isuppz*(2*i-1) through *isuppz*(2*i). This is relevant in the case when the matrix is split. *isuppz* is only accessed when *jobz* = 'V' and *n*>0.

tryrac

On exit, TRUE. *tryrac* is set to .FALSE. if the matrix does not define its eigenvalues to high relative accuracy.

work(1)

On exit, if *info* = 0, then *work*(1) returns the optimal (and minimal) size of *lwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the optimal size of *liwork*.

info

INTEGER.

If = 0, the execution is successful.

If *info* = -i, the i-th parameter had an illegal value.

If *info* = 1, internal error in ?larre occurred,

if *info* = 2, internal error in ?larrv occurred.

?stedc

Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Syntax

```
call sstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call cstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
call zstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
call rstedc(d, e [,z] [,compz] [,info])
call stedc(d, e [,z] [,compz] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band real symmetric or complex Hermitian matrix can also be found if sytrd/hetrd or sptrd/hptrd or sbtrd/hbtrd has been used to reduce this matrix to tridiagonal form.

See also laed0, laed1, laed2, laed3, laed4, laed5, laed6, laed7, laed8, laed9, and laeda used by this function.

Input Parameters

compz

CHARACTER*1. Must be 'N' or 'I' or 'V'.

If *compz* = 'N', the routine computes eigenvalues only.

If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix.

If *compz* = 'V', the routine computes the eigenvalues and eigenvectors of original symmetric/Hermitian matrix. On entry, the array *z* must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.

n

INTEGER. The order of the symmetric tridiagonal matrix ($n \geq 0$).

d, *e*, *rwork*

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays:

d(*) contains the diagonal elements of the tridiagonal matrix.

The dimension of *d* must be at least max(1, *n*).

e(*) contains the subdiagonal elements of the tridiagonal matrix.

The dimension of *e* must be at least max(1, *n*-1).

rwork is a workspace array, its dimension `max(1, lrwork)`.

z, *work*

`REAL` for `sstedc`

`DOUBLE PRECISION` for `dstedc`

`COMPLEX` for `cstedc`

`DOUBLE COMPLEX` for `zstedc`.

Arrays: *z*(*ldz*, *), *work*(*).

If `compz = 'V'`, then, on entry, *z* must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.

The second dimension of *z* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

ldz

`INTEGER`. The leading dimension of *z*. Constraints:

ldz≥ 1 if `compz = 'N'`;

ldz≥ max(1, *n*) if `compz = 'V'` or `'I'`.

lwork

`INTEGER`. The dimension of the array *work*.

For real functions `sstedc` and `dstedc`:

- If `compz = 'N'`or $n \le 1$, *lwork* must be at least 1.
- If `compz = 'V'` and $n > 1$, *lwork* must be at least $1 + 3*n + 2*n*\log_2(n) + 4*n^2$, where $\log_2(n)$ is the smallest integer *k* such that $2^k \ge n$.
- If `compz = 'I'` and $n > 1$ then *lwork* must be at least $1 + 4*n + n^2$

  Note that for `compz = 'I'` or `'V'` and if *n* is less than or equal to the minimum divide size, usually 25, then *lwork* need only be `max(1, 2*(n-1))`.

For complex functions `cstedc` and `zstedc`:

- If `compz = 'N'`or `'I'`, or $n \le 1$, *lwork* must be at least 1.
- If `compz = 'V'` and $n > 1$, *lwork* must be at least n^2.

  Note that for `compz = 'V'`, and if *n* is less than or equal to the minimum divide size, usually 25, then *lwork* need only be 1.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for the required value of *lwork*.

lrwork

`INTEGER`. The dimension of the array *rwork* (used for complex flavors only).

If `compz = 'N'`, or $n \le 1$, *lrwork* must be at least 1.

If *compz* = 'V' and *n* > 1, *lrwork* must be at least (1 + 3**n* + 2**n**$\log_2(n)$ + 4**n*^{2}), where $\log_2(n)$ is the smallest integer *k* such that $2^k \geq n$.

If *compz* = 'I' and *n* > 1, *lrwork* must be at least (1 + 4**n* + 2**n*^{2}).

Note that for *compz* = 'V'or 'I', and if *n* is less than or equal to the minimum divide size, usually 25, then *lrwork* need only be max(1, 2*(*n*-1)).

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for the required value of *lrwork*.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER. The dimension of the array *iwork*.

If *compz* = 'N', or *n* ≤ 1, *liwork* must be at least 1.

If *compz* = 'V' and *n* > 1, *liwork* must be at least (6 + 6**n* + 5**n**$\log_2(n)$), where $\log_2(n)$ is the smallest integer *k* such that $2^k \geq n$.

If *compz* = 'I' and *n* > 1, *liwork* must be at least (3 + 5**n*).

Note that for *compz* = 'V'or 'I', and if *n* is less than or equal to the minimum divide size, usually 25, then *liwork* need only be 1.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for the required value of *liwork*.

Output Parameters

d The *n* eigenvalues in ascending order, unless *info* ≠ 0.

See also *info*.

e On exit, the array is overwritten; see *info*.

z If *info* = 0, then if *compz* = 'V', *z* contains the orthonormal eigenvectors of the original symmetric/Hermitian matrix, and if *compz* = 'I', *z* contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If *compz* = 'N', *z* is not referenced.

work(1) On exit, if *info* = 0, then *work(1)* returns the optimal *lwork*.

rwork(1) On exit, if *info* = 0, then *rwork(1)* returns the optimal *lrwork* (for complex flavors only).

iwork(1) On exit, if *info* = 0, then *iwork(1)* returns the optimal *liwork*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *i*/(*n*+1) through mod(*i*, *n*+1).

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `stedc` interface are the following:

d	Holds the vector of length *n*.
e	Holds the vector of length (*n*-1).
z	Holds the matrix *Z* of size (*n*,*n*).
compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = 'I', if *z* is present, *compz* = 'N', if *z* is omitted. If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted.

Note that two variants of Fortran 95 interface for `stedc` routine are needed because of an ambiguous choice between real and complex cases appear when *z* and *work* are omitted. Thus, the name `rstedc` is used in real cases (single or double precision), and the name `stedc` is used in complex cases (single or double precision).

Application Notes

The required size of workspace arrays must be as follows.

For `sstedc/dstedc`:

If *compz* = 'N' or $n \le 1$ then *lwork* must be at least 1.

If *compz* = 'V' and $n > 1$ then *lwork* must be at least $(1 + 3n + 2n \cdot \log_2 n + 4n^2)$, where $\log_2(n)$ = smallest integer k such that $2^k \ge n$.

If *compz* = 'I' and $n > 1$ then *lwork* must be at least $(1 + 4n + n^2)$.

If *compz* = 'N' or $n \le 1$ then *liwork* must be at least 1.

If *compz* = 'V' and $n > 1$ then *liwork* must be at least $(6 + 6n + 5n \cdot \log_2 n)$.

If *compz* = 'I' and $n > 1$ then *liwork* must be at least $(3 + 5n)$.

For `cstedc/zstedc`:

If *compz* = 'N' or 'I', or $n \le 1$, *lwork* must be at least 1.

If *compz* = 'V' and $n > 1$, *lwork* must be at least n^2.

If *compz* = 'N' or $n \le 1$, *lrwork* must be at least 1.

If *compz* = 'V' and $n > 1$, *lrwork* must be at least $(1 + 3n + 2n \cdot \log_2 n + 4n^2)$, where $\log_2(n)$ = smallest integer k such that $2^k \ge n$.

If *compz* = 'I' and $n > 1$, *lrwork* must be at least$(1 + 4n + 2n^2)$.

The required value of *liwork* for complex flavors is the same as for real flavors.

If *lwork* (or *liwork* or *lrwork*, if supplied) is equal to -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*, *lrwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stegr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
call dstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
call cstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
call zstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
call rstegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
call stegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*.

The spectrum may be computed either completely or partially by specifying either an interval `(vl,vu]` or a range of indices `il:iu` for the desired eigenvalues.

`?stegr` is a compatibility wrapper around the improved stemr routine. See its description for further details.

Note that the *abstol* parameter no longer provides any benefit and hence is no longer used.

See also auxiliary lasq2lasq5, lasq6, used by this routine.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.

If *job* = 'N', then only eigenvalues are computed.

If *job* = 'V', then eigenvalues and eigenvectors are computed.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *w*(*i*) in the half-open interval:

	vl<*w*(*i*)≤*vu*.
	If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
n	INTEGER. The order of the matrix *T* (*n*≥ 0).
d, *e*, *work*	REAL for single precision flavors
	DOUBLE PRECISION for double precision flavors.
	Arrays:
	d(*) contains the diagonal elements of *T*.
	The dimension of *d* must be at least max(1, *n*).
	e(*) contains the subdiagonal elements of *T* in elements 1 to *n*-1; *e*(*n*) need not be set on input, but it is used as a workspace.
	The dimension of *e* must be at least max(1, *n*).
	work(*lwork*) is a workspace array.
vl, *vu*	REAL for single precision flavors
	DOUBLE PRECISION for double precision flavors.
	If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
	Constraint: *vl*< *vu*.
	If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER.
	If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
	Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0.
	If *range* = 'A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for single precision flavors
	DOUBLE PRECISION for double precision flavors.
	Unused. Was the absolute error tolerance for the eigenvalues/eigenvectors in previous versions.
ldz	INTEGER. The leading dimension of the output array *z*. Constraints:
	ldz≥ 1 if *jobz* = 'N';
	ldz≥ max(1, *n*) if *jobz* = 'V'.
lwork	INTEGER.
	The dimension of the array *work*,
	lwork≥max(1, 18**n*) if *jobz* = 'V', and
	lwork≥max(1, 12**n*) if *jobz* = 'N'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* below for details.

iwork INTEGER.

Workspace array, size (*liwork*).

liwork INTEGER.

The dimension of the array *iwork*, *liwork* ≥ max(1, 10*n*) if the eigenvectors are desired, and *liwork* ≥ max(1, 8*n*) if only the eigenvalues are to be computed..

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla. See *Application Notes* below for details.

Output Parameters

d, *e* On exit, *d* and *e* are overwritten.

m INTEGER. The total number of eigenvalues found,

$0 \le m \le n$.

If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, *n*).

The selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*).

z REAL for sstegr

DOUBLE PRECISION for dstegr

COMPLEX for cstegr

DOUBLE COMPLEX for zstegr.

Array *z*(*ldz*, *), the second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', and if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If *jobz* = 'N', then *z* is not referenced.

Note: if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used. Using *n* = *m* is always safe.

isuppz INTEGER.

Array, size at least (2*max(1, *m*)).

	The support of the eigenvectors in *z*, that is the indices indicating the nonzero elements in *z*. The *i*-th computed eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*). This is relevant in the case when the matrix is split. *isuppz* is only accessed when *jobz* = 'V', and *n* > 0.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1*x*, internal error in ?larre occurred, If *info* = 2*x*, internal error in ?larrv occurred. Here the digit *x* = abs(*iinfo*) < 10, where *iinfo* is the non-zero error code returned by ?larre or ?larrv, respectively.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine stegr interface are the following:

d	Holds the vector of length *n*.
e	Holds the vector of length *n*.
w	Holds the vector of length *n*.
z	Holds the matrix *Z* of size (*n*,*m*).
isuppz	Holds the vector of length (2**m*).
vl	Default value for this argument is *vl* = - HUGE (*vl*) where HUGE(*a*) means the largest machine number of the same precision as argument *a*.
vu	Default value for this argument is *vu* = HUGE (*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this argument is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present,

range = 'I', if one of or both *il* and *iu* are present,

range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Note that two variants of Fortran 95 interface for `stegr` routine are needed because of an ambiguous choice between real and complex cases appear when *z* is omitted. Thus, the name `rstegr` is used in real cases (single or double precision), and the name `stegr` is used in complex cases (single or double precision).

Application Notes

`?stegr` works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and `NaN`s. Normal execution of `?stegr` may create `NaN`s and infinities and hence may abort due to a floating point exception in environments which do not conform to the IEEE-754 standard.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?pteqr

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric positive-definite tridiagonal matrix.

Syntax

```
call spteqr(compz, n, d, e, z, ldz, work, info)
call dpteqr(compz, n, d, e, z, ldz, work, info)
call cpteqr(compz, n, d, e, z, ldz, work, info)
call zpteqr(compz, n, d, e, z, ldz, work, info)
call rpteqr(d, e [,z] [,compz] [,info])
call pteqr(d, e [,z] [,compz] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric positive-definite tridiagonal matrix *T*. In other words, the routine can compute the spectral factorization: $T = Z*\Lambda*Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$T*z_i = \lambda_i*z_i$ for $i = 1, 2, \ldots, n$.

(The routine normalizes the eigenvectors so that $||z_i||_2 = 1$.)

You can also use the routine for computing the eigenvalues and eigenvectors of real symmetric (or complex Hermitian) positive-definite matrices A reduced to tridiagonal form T: $A = Q*T*Q^H$. In this case, the spectral factorization is as follows: $A = Q*T*Q^H = (QZ)*\Lambda*(QZ)^H$. Before calling ?pteqr, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	?sytrd, ?orgtr	?hetrd, ?ungtr
packed storage	?sptrd, ?opgtr	?hptrd, ?upgtr
band storage	?sbtrd(*vect*='V')	?hbtrd(*vect*='V')

The routine first factorizes T as $L*D*L^H$ where L is a unit lower bidiagonal matrix, and D is a diagonal matrix. Then it forms the bidiagonal matrix $B = L*D^{1/2}$ and calls ?bdsqr to compute the singular values of B, which are the square roots of the eigenvalues of T.

Input Parameters

compz

CHARACTER*1. Must be 'N' or 'I' or 'V'.

If *compz* = 'N', the routine computes eigenvalues only.

If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T.

If *compz* = 'V', the routine computes the eigenvalues and eigenvectors of A (and the array *z* must contain the matrix Q on entry).

n

INTEGER. The order of the matrix T ($n \geq 0$).

d, *e*, *work*

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays:

d(*) contains the diagonal elements of T.

The size of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of T.

The size of *e* must be at least max(1, *n*-1).

work(*) is a workspace array.

The dimension of *work* must be:

at least 1 if *compz* = 'N';

at least max(1, 4**n*-4) if *compz* = 'V' or 'I'.

z

REAL for spteqr

DOUBLE PRECISION for dpteqr

COMPLEX for cpteqr

DOUBLE COMPLEX for zpteqr.

Array, size (*ldz*,*)

If *compz* = 'N' or 'I', *z* need not be set.

If *compz* = 'V', *z* must contain the orthogonal matrix used in the reduction to tridiagonal form..

The second dimension of *z* must be:

at least 1 if *compz* = 'N';

at least max(1, *n*) if *compz* = 'V' or 'I'.

ldz — INTEGER. The leading dimension of *z*. Constraints:

ldz≥ 1 if *compz* = 'N';

ldz≥ max(1, *n*) if *compz* = 'V' or 'I'.

Output Parameters

d — The *n* eigenvalues in descending order, unless *info* > 0.

See also *info*.

e — On exit, the array is overwritten.

z — If *info* = 0, contains an *n*-by*n* matrix the columns of which are orthonormal eigenvectors. (The *i*-th column corresponds to the *i*-th eigenvalue.)

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, the leading minor of order *i* (and hence *T* itself) is not positive-definite.

If *info* = *n* + *i*, the algorithm for computing singular values failed to converge; *i* off-diagonal elements have not converged to zero.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pteqr interface are the following:

d — Holds the vector of length *n*.

e — Holds the vector of length (*n*-1).

z — Holds the matrix *Z* of size (*n*,*n*).

compz — If omitted, this argument is restored based on the presence of argument *z* as follows:

compz = 'I', if *z* is present,

compz = 'N', if *z* is omitted.

If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted.

Note that two variants of Fortran 95 interface for pteqr routine are needed because of an ambiguous choice between real and complex cases appear when *z* is omitted. Thus, the name rpteqr is used in real cases (single or double precision), and the name pteqr is used in complex cases (single or double precision).

Application Notes

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$|\mu_i - \lambda_i| \leq c(n)*\varepsilon*K*\lambda_i$

where $c(n)$ is a modestly increasing function of n, ε is the machine precision, and $K = ||DTD||_2 * ||(DTD)^{-1}||_2$, D is diagonal with $d_{ii} = t_{ii}^{-1/2}$.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$\theta(u_i, w_i) \leq c(n)\varepsilon K / \min_{i \neq j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|)$.

Here $\min_{i \neq j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|)$ is the *relative gap* between λ_i and the other eigenvalues.

The total number of floating-point operations depends on how rapidly the algorithm converges.

Typically, it is about

$30n^2$ if *compz* = 'N';

$6n^3$ (for complex flavors, $12n^3$) if *compz* = 'V' or 'I'.

?stebz

Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.

Syntax

call sstebz (*range*, *order*, *n*, *vl*, *vu*, *il*, *iu*, *abstol*, *d*, *e*, *m*, *nsplit*, *w*, *iblock*, *isplit*, *work*, *iwork*, *info*)

call dstebz (*range*, *order*, *n*, *vl*, *vu*, *il*, *iu*, *abstol*, *d*, *e*, *m*, *nsplit*, *w*, *iblock*, *isplit*, *work*, *iwork*, *info*)

call stebz(*d*, *e*, *m*, *nsplit*, *w*, *iblock*, *isplit* [, *order*] [,*vl*] [,*vu*] [,*il*] [,*iu*] [,*abstol*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes some (or all) of the eigenvalues of a real symmetric tridiagonal matrix *T* by bisection. The routine searches for zero or negligible off-diagonal elements to see if *T* splits into block-diagonal form $T = \mathrm{diag}(T_1, T_2, \ldots)$. Then it performs bisection on each of the blocks T_i and returns the block index of each computed eigenvalue, so that a subsequent call to stein can also take advantage of the block structure.

See also laebz.

Input Parameters

range

CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval: $vl < w(i) \le vu$.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

order

CHARACTER*1. Must be 'B' or 'E'.

If *order* = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block.

If *order* = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.

n

INTEGER. The order of the matrix *T* ($n \ge 0$).

vl, *vu*

REAL for sstebz

DOUBLE PRECISION for dstebz.

If *range* = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval:

$vl < w(i)) \le vu$.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*

INTEGER. Constraint: $1 \le il \le iu \le n$.

If *range* = 'I', the routine computes eigenvalues $w(i)$ such that $il \le i \le iu$ (assuming that the eigenvalues $w(i)$ are in ascending order).

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for sstebz

DOUBLE PRECISION for dstebz.

The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width *abstol*.

If $abstol \le 0.0$, then the tolerance is taken as eps*|*T*|, where eps is the machine precision, and |*T*| is the 1-norm of the matrix *T*.

d, *e*, *work*

REAL for sstebz

DOUBLE PRECISION for dstebz.

Arrays:

d(*) contains the diagonal elements of *T*.

The size of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.

The size of *e* must be at least max(1, *n*-1).

work(*) is a workspace array.

The dimension of *work* must be at least max(1, 4*n*).

iwork — INTEGER. Workspace.

Array, size at least max(1, 3*n*).

Output Parameters

m — INTEGER. The actual number of eigenvalues found.

nsplit — INTEGER. The number of diagonal blocks detected in *T*.

w — REAL for sstebz

DOUBLE PRECISION for dstebz.

Array, size at least max(1, *n*). The computed eigenvalues, stored in *w*(1) to *w*(*m*).

iblock, *isplit* — INTEGER.

Arrays, size at least max(1, *n*).

A positive value *iblock*(*i*) is the block number of the eigenvalue stored in *w*(*i*) (see also *info*).

The leading *nsplit* elements of *isplit* contain points at which *T* splits into blocks T_i as follows: the block T_1 contains rows/columns 1 to *isplit*(1); the block T_2 contains rows/columns *isplit*(1)+1 to *isplit*(2), and so on.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, for *range* = 'A' or 'V', the algorithm failed to compute some of the required eigenvalues to the desired accuracy; *iblock*(*i*)<0 indicates that the eigenvalue stored in *w*(*i*) failed to converge.

If *info* = 2, for *range* = 'I', the algorithm failed to compute some of the required eigenvalues. Try calling the routine again with *range* = 'A'.

If *info* = 3:

for *range* = 'A' or 'V', same as *info* = 1;

for *range* = 'I', same as *info* = 2.

If *info* = 4, no eigenvalues have been computed. The floating-point arithmetic on the computer is not behaving as expected.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine stebz interface are the following:

d — Holds the vector of length *n*.

e	Holds the vector of length (*n*-1).
w	Holds the vector of length *n*.
iblock	Holds the vector of length *n*.
isplit	Holds the vector of length *n*.
order	Must be 'B' or 'E'. The default value is 'B'.
vl	Default value for this argument is *vl* = - HUGE (*vl*) where HUGE(*a*) means the largest machine number of the same precision as argument *a*.
vu	Default value for this argument is *vu* = HUGE (*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this argument is *abstol* = 0.0_WP.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

The eigenvalues of *T* are computed to high relative accuracy which means that if they vary widely in magnitude, then any small eigenvalues will be computed more accurately than, for example, with the standard *QR* method. However, the reduction to tridiagonal form (prior to calling the routine) may exclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix if its eigenvalues vary widely in magnitude.

?stein

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix.

Syntax

```
call sstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call dstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call cstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call zstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call stein(d, e, w, iblock, isplit, z [,ifailv] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the eigenvectors of a real symmetric tridiagonal matrix *T* corresponding to specified eigenvalues, by inverse iteration. It is designed to be used in particular after the specified eigenvalues have been computed by `?stebz` with `order = 'B'`, but may also be used when the eigenvalues have been computed by other routines.

If you use this routine after `?stebz`, it can take advantage of the block structure by performing inverse iteration on each block T_i separately, which is more efficient than using the whole matrix *T*.

If *T* has been formed by reduction of a full symmetric or Hermitian matrix *A* to tridiagonal form, you can transform eigenvectors of *T* to eigenvectors of *A* by calling `?ormtr` or `?opmtr` (for real flavors) or by calling `?unmtr` or `?upmtr` (for complex flavors).

Input Parameters

n

`INTEGER`. The order of the matrix *T* (*n*≥ 0).

m

`INTEGER`. The number of eigenvectors to be returned.

d, *e*, *w*

`REAL` for single-precision flavors

`DOUBLE PRECISION` for double-precision flavors.

Arrays:

d(*) contains the diagonal elements of *T*.

The size of *d* must be at least max(1, *n*).

e(*) contains the sub-diagonal elements of *T* stored in elements 1 to *n*-1

The size of *e* must be at least max(1, *n*-1).

w(*) contains the eigenvalues of *T*, stored in `w(1)` to `w(m)` (as returned by stebz). Eigenvalues of T_1 must be supplied first, in non-decreasing order; then those of T_2, again in non-decreasing order, and so on. Constraint:

if *iblock*(*i*) = *iblock*(*i*+1), *w*(*i*) ≤*w*(*i*+1).

The size of *w* must be at least max(1, *n*).

iblock, *isplit*

`INTEGER`.

Arrays, size at least max(1, *n*). The arrays *iblock* and *isplit*, as returned by `?stebz` with `order = 'B'`.

If you did not call `?stebz` with `order = 'B'`, set all elements of *iblock* to 1, and *isplit*(1) to *n*.)

ldz

`INTEGER`. The leading dimension of the output array *z*; *ldz*≥ max(1, *n*).

work

`REAL` for single-precision flavors

`DOUBLE PRECISION` for double-precision flavors.

Workspace array, size at least max(1, 5*n*).

iwork

`INTEGER`.

Workspace array, size at least max(1, *n*).

Output Parameters

z

`REAL` for `sstein`

`DOUBLE PRECISION` for `dstein`

COMPLEX for cstein

DOUBLE COMPLEX for zstein.

Array, size (*ldz*, *).

If *info* = 0, *z* contains an *n*-by-*n* matrix the columns of which are orthonormal eigenvectors. (The *i*-th column corresponds to the *i*th eigenvalue.)

ifailv INTEGER.

Array, size at least max(1, *m*).

If *info* = *i* > 0, the first *i* elements of *ifailv* contain the indices of any eigenvectors that failed to converge.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, then *i* eigenvectors (as indicated by the parameter *ifailv*) each failed to converge in 5 iterations. The current iterates are stored in the corresponding columns of the array *z*.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine stein interface are the following:

d Holds the vector of length *n*.

e Holds the vector of length *n*.

w Holds the vector of length *n*.

iblock Holds the vector of length *n*.

isplit Holds the vector of length *n*.

z Holds the matrix *Z* of size (*n*,*m*).

ifailv Holds the vector of length (*m*).

Application Notes

Each computed eigenvector z_i is an exact eigenvector of a matrix $T+E_i$, where $||E_i||_2 = O(\varepsilon)*||T||_2$. However, a set of eigenvectors computed by this routine may not be orthogonal to so high a degree of accuracy as those computed by ?steqr.

?disna

Computes the reciprocal condition numbers for the eigenvectors of a symmetric/ Hermitian matrix or for the left or right singular vectors of a general matrix.

Syntax

```
call sdisna(job, m, n, d, sep, info)
```

```
call ddisna(job, m, n, d, sep, info)
call disna(d, sep [,job] [,minmn] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general *m*-by-*n* matrix.

The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the *i*-th computed vector is given by

`?lamch('E')*(anorm/sep(i))`

where *anorm* = $||A||_2$ = max(|d(j)|). *sep*(*i*) is not allowed to be smaller than `slamch('E')*anorm` in order to limit the size of the error bound.

`?disna` may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

Input Parameters

job

CHARACTER*1. Must be 'E','L', or 'R'. Specifies for which problem the reciprocal condition numbers should be computed:

job = 'E': for the eigenvectors of a symmetric/Hermitian matrix;

job = 'L': for the left singular vectors of a general matrix;

job = 'R': for the right singular vectors of a general matrix.

m

INTEGER. The number of rows of the matrix ($m \geq 0$).

n

INTEGER.

If *job* = 'L', or 'R', the number of columns of the matrix ($n \geq 0$). Ignored if *job* = 'E'.

d

REAL for sdisna

DOUBLE PRECISION for ddisna.

Array, dimension at least max(1,*m*) if *job* = 'E', and at least max(1, min(*m*,*n*)) if *job* = 'L' or 'R'.

This array must contain the eigenvalues (if *job* = 'E') or singular values (if *job* = 'L' or 'R') of the matrix, in either increasing or decreasing order.

If singular values, they must be non-negative.

Output Parameters

sep

REAL for sdisna

DOUBLE PRECISION for ddisna.

Array, dimension at least max(1,*m*) if *job* = 'E', and at least max(1, min(*m*,*n*)) if *job* = 'L' or 'R'. The reciprocal condition numbers of the vectors.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine disna interface are the following:

d Holds the vector of length min(*m*,*n*).

sep Holds the vector of length min(*m*,*n*).

job Must be 'E', 'L', or 'R'. The default value is 'E'.

minmn Indicates which of the values *m* or *n* is smaller. Must be either 'M' or 'N', the default is 'M'.

If *job* = 'E', this argument is superfluous, If *job* = 'L' or 'R', this argument is used by the routine.

Generalized Symmetric-Definite Eigenvalue Problems: LAPACK Computational Routines

Generalized symmetric-definite eigenvalue problems are as follows: find the eigenvalues λ and the corresponding eigenvectors *z* that satisfy one of these equations:

$Az = \lambda Bz$, $ABz = \lambda z$, or $BAz = \lambda z$,

where *A* is an *n*-by-*n* symmetric or Hermitian matrix, and *B* is an *n*-by-*n* symmetric positive-definite or Hermitian positive-definite matrix.

In these problems, there exist *n* real eigenvectors corresponding to real eigenvalues (even for complex Hermitian matrices *A* and *B*).

Routines described in this topic allow you to reduce the above generalized problems to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling LAPACK routines described earlier in this chapter (see Symmetric Eigenvalue Problems).

Different routines allow the matrices to be stored either conventionally or in packed storage. Prior to reduction, the positive-definite matrix *B* must first be factorized using either potrf or pptrf.

The reduction routine for the banded matrices *A* and *B* uses a split Cholesky factorization for which a specific routine pbstf is provided. This refinement halves the amount of work required to form matrix *C*.

Table "Computational Routines for Reducing Generalized Eigenproblems to Standard Problems" lists LAPACK routines that can be used to solve generalized symmetric-definite eigenvalue problems. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Matrix type	Reduce to standard problems (full storage)	Reduce to standard problems (packed storage)	Reduce to standard problems (band matrices)	Factorize band matrix
real symmetric matrices	sygst	spgst	sbgst	pbstf
complex Hermitian matrices	hegst	hpgst	hbgst	pbstf

?sygst
Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

```
call ssygst(itype, uplo, n, a, lda, b, ldb, info)
call dsygst(itype, uplo, n, a, lda, b, ldb, info)
call sygst(a, b [,itype] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces real symmetric-definite generalized eigenproblems

$A*z = \lambda*B*z$, $A*B*z = \lambda*z$, or $B*A*z = \lambda*z$

to the standard form $C*y = \lambda*y$. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call ?potrf to compute the Cholesky factorization: $B = U^T*U$ or $B = L*L^T$.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3.

If $itype = 1$, the generalized eigenproblem is $A*z = lambda*B*z$

for $uplo$ = 'U': $C = inv(U^T)*A*inv(U)$, $z = inv(U)*y$;

for $uplo$ = 'L': $C = inv(L)*A*inv(L^T)$, $z = inv(L^T)*y$.

If $itype = 2$, the generalized eigenproblem is $A*B*z = lambda*z$

for $uplo$ = 'U': $C = U*A*U^T$, $z = inv(U)*y$;

for $uplo$ = 'L': $C = L^T*A*L$, $z = inv(L^T)*y$.

If $itype = 3$, the generalized eigenproblem is $B*A*z = lambda*z$

for $uplo$ = 'U': $C = U*A*U^T$, $z = U^T*y$;

for $uplo$ = 'L': $C = L^T*A*L$, $z = L*y$.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', the array *a* stores the upper triangle of *A*; you must supply *B* in the factored form $B = U^T*U$.

If *uplo* = 'L', the array *a* stores the lower triangle of *A*; you must supply *B* in the factored form $B = L*L^T$.

n — INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, *b* — REAL for ssygst

DOUBLE PRECISION for dsygst.

Arrays:

a(*lda*,*) contains the upper or lower triangle of *A*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the Cholesky-factored matrix *B*:

$B = U^T*U$ or $B = L*L^T$ (as returned by ?potrf).

The second dimension of *b* must be at least max(1, *n*).

lda — INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The leading dimension of *b*; at least max(1, *n*).

Output Parameters

a — The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sygst interface are the following:

a — Holds the matrix *A* of size (*n*,*n*).

b — Holds the matrix *B* of size (*n*,*n*).

itype — Must be 1, 2, or 3. The default value is 1.

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by inv(*B*) (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?hegst
Reduces a complex Hermitian positive-definite generalized eigenvalue problem to the standard form.

Syntax

```
call chegst(itype, uplo, n, a, lda, b, ldb, info)
call zhegst(itype, uplo, n, a, lda, b, ldb, info)
call hegst(a, b [,itype] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a complex Hermitian positive-definite generalized eigenvalue problem to standard form.

itype	**Problem**	**Result**
1	$A*x = \lambda*B*x$	*A* overwritten by $\mathrm{inv}(U^H)*A*\mathrm{inv}(U)$ or $\mathrm{inv}(L)*A*\mathrm{inv}(L^H)$
2	$A*B*x = \lambda*x$	*A* overwritten by $U*A*U^H$ or L^H*A*L
3	$B*A*x = \lambda*x$	

Before calling this routine, you must call ?potrf to compute the Cholesky factorization: $B = U^H*U$ or $B = L*L^H$.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3.

If *itype* = 1, the generalized eigenproblem is *A*z = lambda*B*z*

for *uplo* = 'U': $C = (U^H)^{-1}*A*U^{-1}$;

for *uplo* = 'L': $C = L^{-1}*A*(L^H)^{-1}$.

If *itype* = 2, the generalized eigenproblem is *A*B*z = lambda*z*

for *uplo* = 'U': $C = U*A*U^H$;

for *uplo* = 'L': $C = L^H*A*L$.

If *itype* = 3, the generalized eigenproblem is *B*A*z = lambda*z*

for *uplo* = 'U': $C = U*A*U^H$;

for *uplo* = 'L': $C = L^H*A*L$.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', the array *a* stores the upper triangle of *A*; you must supply *B* in the factored form $B = U^H*U$.

If *uplo* = 'L', the array *a* stores the lower triangle of *A*; you must supply *B* in the factored form $B = L*L^H$.

n

INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, *b*	`COMPLEX` for `chegst` `DOUBLE COMPLEX` for `zhegst`. Arrays: *a*(*lda*,*) contains the upper or lower triangle of *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the Cholesky-factored matrix *B*: $B = U^H*U$ or $B = L*L^H$ (as returned by `?potrf`). The second dimension of *b* must be at least max(1, *n*).
lda	`INTEGER`. The leading dimension of *a*; at least max(1, *n*).
ldb	`INTEGER`. The leading dimension of *b*; at least max(1, *n*).

Output Parameters

a	The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hegst` interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by B^{-1} (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?spgst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

```
call sspgst(itype, uplo, n, ap, bp, info)
```

```
call dspgst(itype, uplo, n, ap, bp, info)
call spgst(ap, bp [,itype] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces real symmetric-definite generalized eigenproblems

$A*x = \lambda*B*x$, $A*B*x = \lambda*x$, or $B*A*x = \lambda*x$

to the standard form $C*y = \lambda*y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call ?pptrf to compute the Cholesky factorization: $B = U^T*U$ or $B = L*L^T$.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3.

If *itype* = 1, the generalized eigenproblem is $A*z = lambda*B*z$

for *uplo* = 'U': $C = \text{inv}(U^T)*A*\text{inv}(U)$, $z = \text{inv}(U)*y$;

for *uplo* = 'L': $C = \text{inv}(L)*A*\text{inv}(L^T)$, $z = \text{inv}(L^T)*y$.

If *itype* = 2, the generalized eigenproblem is $A*B*z = lambda*z$

for *uplo* = 'U': $C = U*A*U^T$, $z = \text{inv}(U)*y$;

for *uplo* = 'L': $C = L^T*A*L$, $z = \text{inv}(L^T)*y$.

If *itype* = 3, the generalized eigenproblem is $B*A*z = lambda*z$

for *uplo* = 'U': $C = U*A*U^T$, $z = U^T*y$;

for *uplo* = 'L': $C = L^T*A*L$, $z = L*y$.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangle of *A*;

you must supply *B* in the factored form $B = U^T*U$.

If *uplo* = 'L', *ap* stores the packed lower triangle of *A*;

you must supply *B* in the factored form $B = L*L^T$.

n

INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, *bp*

REAL for sspgst

DOUBLE PRECISION for dspgst.

Arrays:

ap(*) contains the packed upper or lower triangle of *A*.

The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed Cholesky factor of *B* (as returned by ?pptrf with the same *uplo* value).

The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).

Output Parameters

ap	The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine spgst interface are the following:

ap	Holds the array *A* of size (*n**(*n*+1)/2).
bp	Holds the array *B* of size (*n**(*n*+1)/2).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by inv(*B*) (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?hpgst

Reduces a generalized eigenvalue problem with a Hermitian matrix to a standard eigenvalue problem using packed storage.

Syntax

```
call chpgst(itype, uplo, n, ap, bp, info)
call zhpgst(itype, uplo, n, ap, bp, info)
call hpgst(ap, bp [,itype] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces generalized eigenproblems with Hermitian matrices

*A***z* = λ**B***z*, *A***B***z* = λ**z*, or *B***A***z* = λ**z*.

to standard eigenproblems $C*y = \lambda*y$, using packed matrix storage. Here A is a complex Hermitian matrix, and B is a complex Hermitian positive-definite matrix. Before calling this routine, you must call ?pptrf to compute the Cholesky factorization: $B = U^H*U$ or $B = L*L^H$.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3.

If *itype* = 1, the generalized eigenproblem is $A*z = lambda*B*z$

for *uplo* = 'U': $C = \text{inv}(U^H)*A*\text{inv}(U)$, $z = \text{inv}(U)*y$;

for *uplo* = 'L': $C = \text{inv}(L)*A*\text{inv}(L^H)$, $z = \text{inv}(L^H)*y$.

If *itype* = 2, the generalized eigenproblem is $A*B*z = lambda*z$

for *uplo* = 'U': $C = U*A*U^H$, $z = \text{inv}(U)*y$;

for *uplo* = 'L': $C = L^H*A*L$, $z = \text{inv}(L^H)*y$.

If *itype* = 3, the generalized eigenproblem is $B*A*z = lambda*z$

for *uplo* = 'U': $C = U*A*U^H$, $z = U^H*y$;

for *uplo* = 'L': $C = L^H*A*L$, $z = L*y$.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangle of *A*; you must supply *B* in the factored form $B = U^H*U$.

If *uplo* = 'L', *ap* stores the packed lower triangle of *A*; you must supply *B* in the factored form $B = L*L^H$.

n

INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, *bp*

COMPLEX for chpgstDOUBLE COMPLEX for zhpgst.

Arrays:

ap(*) contains the packed upper or lower triangle of *A*.

The dimension of *a* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed Cholesky factor of *B* (as returned by ?pptrf with the same *uplo* value).

The dimension of *b* must be at least max(1, *n**(*n*+1)/2).

Output Parameters

ap

The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hpgst` interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
bp	Holds the array *B* of size (*n*(n+1)/2*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by inv(*B*) (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?sbgst

Reduces a real symmetric-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by `?pbstf`.

Syntax

```
call ssbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
call dsbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
call sbgst(ab, bb [,x] [,uplo] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

To reduce the real symmetric-definite generalized eigenproblem $A*z = \lambda*B*z$ to the standard form $C*y=\lambda*y$, where *A*, *B* and *C* are banded, this routine must be preceded by a call to pbstf, which computes the split Cholesky factorization of the positive-definite matrix *B*: $B=S^T*S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites *A* with $C = X^T*A*X$, where $X = \text{inv}(S)*Q$ and *Q* is an orthogonal matrix chosen (implicitly) to preserve the bandwidth of *A*. The routine also has an option to allow the accumulation of *X*, and then, if *z* is an eigenvector of *C*, $X*z$ is an eigenvector of the original system.

Input Parameters

vect	`CHARACTER*1`. Must be `'N'` or `'V'`. If *vect* = `'N'`, then matrix *X* is not returned; If *vect* = `'V'`, then matrix *X* is returned.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, *ab* stores the upper triangular part of *A*. If *uplo* = `'L'`, *ab* stores the lower triangular part of *A*.

n INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

ka INTEGER. The number of super- or sub-diagonals in *A* (*ka*≥ 0).

kb INTEGER. The number of super- or sub-diagonals in *B* (*ka*≥*kb*≥ 0).

ab, *bb*, *work* REAL for ssbgst

DOUBLE PRECISION for dsbgst

ab(*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of the array *ab* must be at least max(1, *n*).

bb(*ldbb*,*) is an array containing the banded split Cholesky factor of *B* as specified by *uplo*, *n* and *kb* and returned by pbstf/pbstf.

The second dimension of the array *bb* must be at least max(1, *n*).

work(*) is a workspace array, dimension at least max(1, 2**n*)

ldab INTEGER. The leading dimension of the array *ab*; must be at least *ka*+1.

ldbb INTEGER. The leading dimension of the array *bb*; must be at least *kb*+1.

ldx The leading dimension of the output array x. Constraints: if *vect* = 'N', then *ldx*≥ 1;

if *vect* = 'V', then *ldx*≥ max(1, *n*).

Output Parameters

ab On exit, this array is overwritten by the upper or lower triangle of *C* as specified by *uplo*.

x REAL for ssbgst

DOUBLE PRECISION for dsbgst

Array.

If *vect* = 'V', then *x*(*ldx*,*) contains the *n*-by-*n* matrix *X* = inv(*S*)**Q*.

If *vect* = 'N', then *x* is not referenced.

The second dimension of x must be:

at least max(1, *n*), if *vect* = 'V';

at least 1, if *vect* = 'N'.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sbgst` interface are the following:

ab	Holds the array *A* of size (*ka+1,n*).
bb	Holds the array *B* of size (*kb+1,n*).
x	Holds the matrix *X* of size (*n,n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
vect	Restored based on the presence of the argument *x* as follows: *vect* = `'V'`, if *x* is present, *vect* = `'N'`, if *x* is omitted.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by `inv(B)`. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

If *ka* and *kb* are much less than *n* then the total number of floating-point operations is approximately $6n^2*kb$, when *vect* = `'N'`. Additional $(3/2)n^3*(kb/ka)$ operations are required when *vect* = `'V'`.

?hbgst

Reduces a complex Hermitian positive-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by `?pbstf`.

Syntax

```
call chbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork, info)
call zhbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork, info)
call hbgst(ab, bb [,x] [,uplo] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

To reduce the complex Hermitian positive-definite generalized eigenproblem $A*z = \lambda*B*z$ to the standard form $C*x = \lambda*y$, where *A*, *B* and *C* are banded, this routine must be preceded by a call to pbstf/pbstf, which computes the split Cholesky factorization of the positive-definite matrix *B*: $B = S^H*S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites *A* with $C = X^H*A*X$, where $X = \text{inv}(S)*Q$, and *Q* is a unitary matrix chosen (implicitly) to preserve the bandwidth of *A*. The routine also has an option to allow the accumulation of *X*, and then, if *z* is an eigenvector of *C*, *X*z* is an eigenvector of the original system.

Input Parameters

vect — CHARACTER*1. Must be 'N' or 'V'.

If *vect* = 'N', then matrix *X* is not returned;

If *vect* = 'V', then matrix *X* is returned.

uplo — CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ab* stores the upper triangular part of *A*.

If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n — INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

ka — INTEGER. The number of super- or sub-diagonals in *A*

(*ka*≥ 0).

kb — INTEGER. The number of super- or sub-diagonals in *B*

(*ka*≥*kb*≥ 0).

ab, *bb*, *work* — COMPLEX for chbgstDOUBLE COMPLEX for zhbgst

ab(*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of the array *ab* must be at least max(1, *n*).

bb(*ldbb*,*) is an array containing the banded split Cholesky factor of *B* as specified by *uplo*, *n* and *kb* and returned by pbstf/pbstf.

The second dimension of the array *bb* must be at least max(1, *n*).

work(*) is a workspace array, dimension at least max(1, *n*)

ldab — INTEGER. The leading dimension of the array *ab*; must be at least *ka*+1.

ldbb — INTEGER. The leading dimension of the array *bb*; must be at least *kb*+1.

ldx — The leading dimension of the output array x. Constraints:

if *vect* = 'N', then *ldx*≥ 1;

if *vect* = 'V', then *ldx*≥ max(1, *n*).

rwork — REAL for chbgst

DOUBLE PRECISION for zhbgst

Workspace array, dimension at least max(1, *n*)

Output Parameters

ab — On exit, this array is overwritten by the upper or lower triangle of *C* as specified by *uplo*.

x — COMPLEX for chbgst

DOUBLE COMPLEX for zhbgst

Array.

If *vect* = 'V', then *x*(*ldx*,*) contains the *n*-by-*n* matrix *X* = inv(*S*)*Q*.

If *vect* = 'N', then *x* is not referenced.

The second dimension of x must be:

at least max(1, *n*), if *vect* = 'V';

at least 1, if *vect* = 'N'.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hbgst interface are the following:

ab Holds the array *A* of size (*ka+1,n*).

bb Holds the array *B* of size (*kb+1,n*).

x Holds the matrix *X* of size (*n,n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

vect Restored based on the presence of the argument *x* as follows: *vect* = 'V', if *x* is present, *vect* = 'N', if *x* is omitted.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by inv(*B*). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion. The total number of floating-point operations is approximately $20n^2*kb$, when *vect* = 'N'. Additional $5n^3*(kb/ka)$ operations are required when *vect* = 'V'. All these estimates assume that both *ka* and *kb* are much less than *n*.

?pbstf

Computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite banded matrix used in ?sbgst/?hbgst *.*

Syntax

```
call spbstf(uplo, n, kb, bb, ldbb, info)
call dpbstf(uplo, n, kb, bb, ldbb, info)
call cpbstf(uplo, n, kb, bb, ldbb, info)
call zpbstf(uplo, n, kb, bb, ldbb, info)
call pbstf(bb [, uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite band matrix *B*. It is to be used in conjunction with sbgst/hbgst.

The factorization has the form $B = S^T*S$ (or $B = S^H*S$ for complex flavors), where *S* is a band matrix of the same bandwidth as *B* and the following structure: S is upper triangular in the first (*n*+*kb*)/2 rows and lower triangular in the remaining rows.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *bb* stores the upper triangular part of *B*.

If *uplo* = 'L', *bb* stores the lower triangular part of *B*.

n — INTEGER. The order of the matrix *B* (*n* ≥ 0).

kb — INTEGER. The number of super- or sub-diagonals in *B* (*kb* ≥ 0).

bb — REAL for spbstf

DOUBLE PRECISION for dpbstf

COMPLEX for cpbstf

DOUBLE COMPLEX for zpbstf.

bb(*ldbb*,*) is an array containing either upper or lower triangular part of the matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least max(1, *n*).

ldbb — INTEGER. The leading dimension of *bb*; must be at least *kb*+1.

Output Parameters

bb — On exit, this array is overwritten by the elements of the split Cholesky factor *S*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, then the factorization could not be completed, because the updated element b_{ii} would be the square root of a negative number; hence the matrix *B* is not positive-definite.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine pbstf interface are the following:

bb — Holds the array *B* of size (*kb*+*1*,*n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed factor *S* is the exact factor of a perturbed matrix $B + E$, where

$$|E| \le c(kb+1)\varepsilon |S^H||S|, \quad |e_{ij}| \le c(kb+1)\varepsilon \sqrt{b_{ii}b_{jj}}$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

The total number of floating-point operations for real flavors is approximately $n(kb+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that kb is much less than n.

After calling this routine, you can call sbgst/hbgst to solve the generalized eigenproblem $Az = \lambda Bz$, where A and B are banded and B is positive-definite.

Nonsymmetric Eigenvalue Problems: LAPACK Computational Routines

This topic describes LAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

A *nonsymmetric eigenvalue problem* is as follows: given a nonsymmetric (or non-Hermitian) matrix *A*, find the *eigenvalues*λ and the corresponding *eigenvectors*z that satisfy the equation

$Az = \lambda z$ (right eigenvectors z)

or the equation

$z^H A = \lambda z^H$ (left eigenvectors z).

Nonsymmetric eigenvalue problems have the following properties:

- The number of eigenvectors may be less than the matrix order (but is not less than the number of *distinct eigenvalues* of *A*).
- Eigenvalues may be complex even for a real matrix *A*.
- If a real nonsymmetric matrix has a complex eigenvalue $a+bi$ corresponding to an eigenvector z, then $a-bi$ is also an eigenvalue. The eigenvalue $a-bi$ corresponds to the eigenvector whose elements are complex conjugate to the elements of z.

To solve a nonsymmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained. Table "Computational Routines for Solving Nonsymmetric Eigenvalue Problems" lists LAPACK routines to reduce the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$ as well as routines to solve eigenvalue problems with Hessenberg matrices, forming the Schur factorization of such matrices and computing the corresponding condition numbers. The corresponding routine names in the Fortran 95 interface are without the first symbol.

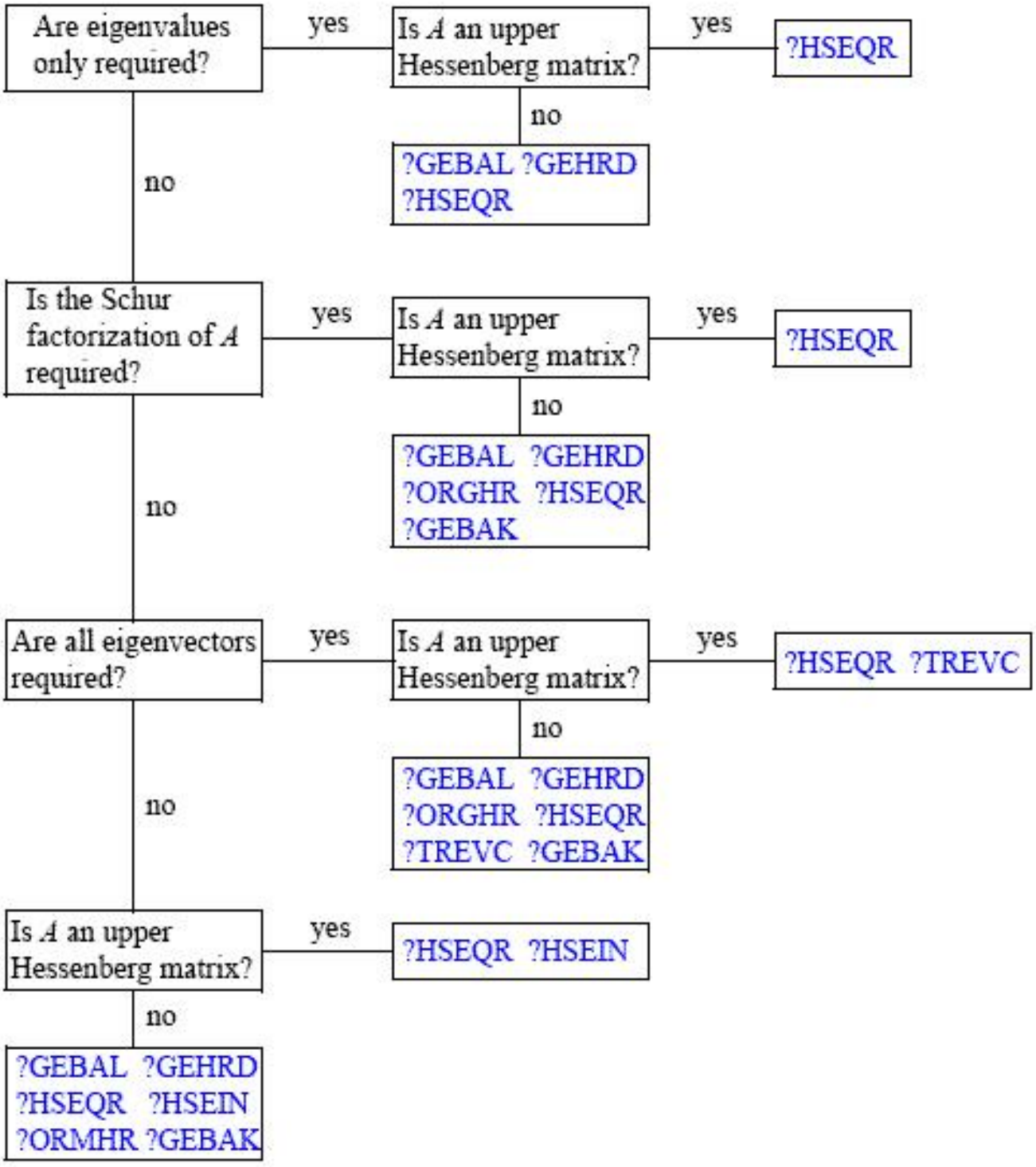
The decision tree in Figure "Decision Tree: Real Nonsymmetric Eigenvalue Problems" helps you choose the right routine or sequence of routines for an eigenvalue problem with a real nonsymmetric matrix. If you need to solve an eigenvalue problem with a complex non-Hermitian matrix, use the decision tree shown in Figure "Decision Tree: Complex Non-Hermitian Eigenvalue Problems".

Computational Routines for Solving Nonsymmetric Eigenvalue Problems

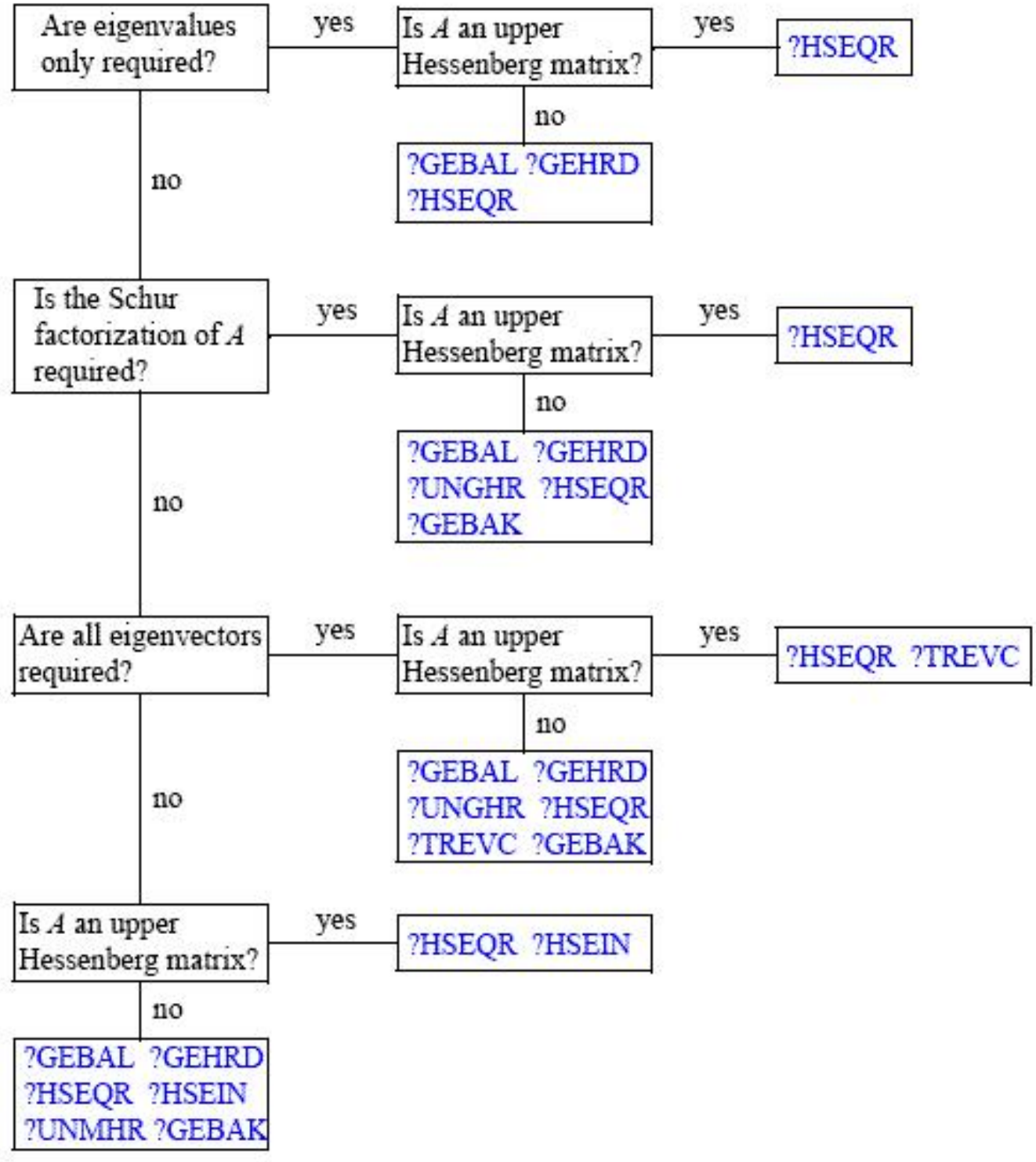
Operation performed	Routines for real matrices	Routines for complex matrices
Reduce to Hessenberg form $A = QHQ^H$	?gehrd,	?gehrd
Generate the matrix Q	?orghr	?unghr
Apply the matrix Q	?ormhr	?unmhr

Operation performed	Routines for real matrices	Routines for complex matrices
Balance matrix	?gebal	?gebal
Transform eigenvectors of balanced matrix to those of the original matrix	?gebak	?gebak
Find eigenvalues and Schur factorization (QR algorithm)	?hseqr	?hseqr
Find eigenvectors from Hessenberg form (inverse iteration)	?hsein	?hsein
Find eigenvectors from Schur factorization	?trevc	?trevc
Estimate sensitivities of eigenvalues and eigenvectors	?trsna	?trsna
Reorder Schur factorization	?trexc	?trexc
Reorder Schur factorization, find the invariant subspace and estimate sensitivities	?trsen	?trsen
Solves Sylvester's equation.	?trsyl	?trsyl

Decision Tree: Real Nonsymmetric Eigenvalue Problems



Decision Tree: Complex Non-Hermitian Eigenvalue Problems



?gehrd
Reduces a general matrix to upper Hessenberg form.

Syntax

```
call sgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
```

```
call dgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call cgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call zgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call gehrd(a [, tau] [,ilo] [,ihi] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a general matrix *A* to upper Hessenberg form *H* by an orthogonal or unitary similarity transformation $A = Q*H*Q^H$. Here *H* has real subdiagonal elements.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of *elementary reflectors*. Routines are provided to work with *Q* in this representation.

Input Parameters

n	INTEGER. The order of the matrix *A* ($n \geq 0$).
ilo, *ihi*	INTEGER. If *A* is an output by ?gebal, then *ilo* and *ihi* must contain the values returned by that routine. Otherwise $ilo = 1$ and $ihi = n$. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)
a, *work*	REAL for sgehrd DOUBLE PRECISION for dgehrd COMPLEX for cgehrd DOUBLE COMPLEX for zgehrd. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* (*lwork*) is a workspace array.
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array; at least max(1, *n*). If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	The elements on and above the subdiagonal contain the upper Hessenberg matrix *H*. The subdiagonal elements of *H* are real. The elements below the subdiagonal, with the array tau, represent the orthogonal matrix *Q* as a product of n elementary reflectors.

tau	`REAL` for `sgehrd`
	`DOUBLE PRECISION` for `dgehrd`
	`COMPLEX` for `cgehrd`
	`DOUBLE COMPLEX` for `zgehrd`.
	Array, size at least max (1, *n*-1).
	Contains scalars that define elementary reflectors for the matrix *Q*.

work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gehrd` interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*-1).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.

Application Notes

For better performance, try using *lwork* = *n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Hessenberg matrix *H* is exactly similar to a nearby matrix $A + E$, where $||E||_2 < c(n)\varepsilon||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is $(2/3)*(ihi - ilo)^2(2ihi + 2ilo + 3n)$; for complex flavors it is 4 times greater.

?orghr

Generates the real orthogonal matrix Q determined by ?gehrd*.*

Syntax

```
call sorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call dorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call orghr(a, tau [,ilo] [,ihi] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine explicitly generates the orthogonal matrix *Q* that has been determined by a preceding call to sgehrd/dgehrd. (The routine ?gehrd reduces a real general matrix *A* to upper Hessenberg form *H* by an orthogonal similarity transformation, $A = Q*H*Q^T$, and represents the matrix *Q* as a product of *ihi*–*ilo* *elementary reflectors*. Here *ilo* and *ihi* are values determined by sgebal/dgebal when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

The matrix *Q* generated by ?orghr has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns *ilo* to *ihi*.

Input Parameters

n — INTEGER. The order of the matrix *Q* ($n \geq 0$).

ilo, *ihi* — INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)

a, *tau*, *work*	REAL for sorghr DOUBLE PRECISION for dorghr Arrays: *a*(*lda*,*) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. The second dimension of *a* must be at least max(1, *n*). *tau*(*) contains further details of the elementary reflectors, as returned by ?gehrd. The dimension of *tau* must be at least max (1, *n*-1). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array; *lwork*≥ max(1, *ihi*-*ilo*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *n*-by-*n* orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine orghr interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*-1).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.

Application Notes

For better performance, try using *lwork* =(*ihi*-*ilo*)***blocksize* where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)(ihi-ilo)^3$.

The complex counterpart of this routine is unghr.

?ormhr

Multiplies an arbitrary real matrix C by the real orthogonal matrix Q determined by ?gehrd.

Syntax

```
call sormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call dormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call ormhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a matrix *C* by the orthogonal matrix *Q* that has been determined by a preceding call to sgehrd/dgehrd. (The routine ?gehrd reduces a real general matrix *A* to upper Hessenberg form *H* by an orthogonal similarity transformation, $A = Q*H*Q^T$, and represents the matrix *Q* as a product of *ihi-iloelementary reflectors*. Here *ilo* and *ihi* are values determined by sgebal/dgebal when balancing the matrix;if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

With ?ormhr, you can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$, overwriting the result on *C* (which may be any real rectangular matrix).

A common application of ?ormhr is to transform a matrix *V* of eigenvectors of *H* to the matrix *QV* of eigenvectors of *A*.

Input Parameters

side — CHARACTER*1. Must be 'L' or 'R'.

If *side*= 'L', then the routine forms $Q*C$ or Q^T*C.

If *side*= 'R', then the routine forms $C*Q$ or $C*Q^T$.

trans — CHARACTER*1. Must be 'N' or 'T'.

If *trans*= 'N', then *Q* is applied to *C*.

If *trans*= 'T', then Q^T is applied to *C*.

m — INTEGER. The number of rows in *C* (*m*≥ 0).

n — INTEGER. The number of columns in *C* (*n*≥ 0).

ilo, *ihi* — INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd.

If *m* > 0 and *side* = 'L', then 1 ≤*ilo*≤*ihi*≤*m*.

If *m* = 0 and *side* = 'L', then *ilo* = 1 and *ihi* = 0.

If *n* > 0 and *side* = 'R', then 1 ≤*ilo*≤*ihi*≤*n*.

If *n* = 0 and *side* = 'R', then *ilo* = 1 and *ihi* = 0.

a, *tau*, *c*, *work* — REAL for sormhr

DOUBLE PRECISION for dormhr

Arrays:

a(*lda*,*) contains details of the vectors which define the *elementary reflectors*, as returned by ?gehrd.

The second dimension of *a* must be at least max(1, *m*) if *side* = 'L' and at least max(1, *n*) if *side* = 'R'.

tau(*) contains further details of the *elementary reflectors*, as returned by ?gehrd .

The dimension of *tau* must be at least max (1, *m*-1) if *side* = 'L' and at least max (1, *n*-1) if *side* = 'R'.

c(*ldc*,*) contains the *m* by *n* matrix *C*.

The second dimension of *c* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; at least max(1, *m*) if *side* = 'L' and at least max (1, *n*) if *side* = 'R'.

ldc — INTEGER. The leading dimension of *c*; at least max(1, *m*) .

lwork — INTEGER. The size of the *work* array.

If *side* = 'L', *lwork*≥ max(1, *n*).

If *side* = 'R', *lwork*≥ max(1, *m*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c	*C* is overwritten by product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ as specified by *side* and *trans*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ormhr interface are the following:

a	Holds the matrix *A* of size (*r*,*r*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*r*-1).
c	Holds the matrix *C* of size (*m*,*n*).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, *lwork* should be at least *n***blocksize* if *side* = 'L' and at least *m***blocksize* if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $||E||_2 = O(\varepsilon)*||C||_2$, where ε is the machine precision.

The approximate number of floating-point operations is

$2n(ihi-ilo)^2$ if *side* = 'L';

$2m(ihi-ilo)^2$ if *side* = 'R'.

The complex counterpart of this routine is unmhr.

?unghr
Generates the complex unitary matrix Q determined by ?gehrd*.*

Syntax

```
call cunghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call zunghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call unghr(a, tau [,ilo] [,ihi] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine is intended to be used following a call to cgehrd/zgehrd, which reduces a complex matrix *A* to upper Hessenberg form *H* by a unitary similarity transformation: $A = Q*H*Q^H$. ?gehrd represents the matrix *Q* as a product of *ihi-iloelementary reflectors*. Here *ilo* and *ihi* are values determined by cgebal/zgebal when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.

Use the routine unghr to generate *Q* explicitly as a square matrix. The matrix *Q* has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns *ilo* to *ihi*.

Input Parameters

n	INTEGER. The order of the matrix *Q* (*n*≥ 0).
ilo, *ihi*	INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd . (If *n* > 0, then 1 ≤*ilo*≤*ihi*≤*n*. If *n* = 0, then *ilo* = 1 and *ihi* = 0.)
a, *tau*, *work*	COMPLEX for cunghr DOUBLE COMPLEX for zunghr. Arrays: *a*(*lda*,*) contains details of the vectors which define the *elementary reflectors*, as returned by ?gehrd. The second dimension of *a* must be at least max(1, *n*). *tau*(*) contains further details of the *elementary reflectors*, as returned by ?gehrd . The dimension of *tau* must be at least max (1, *n*-1). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array; *lwork*≥ max(1, *ihi*-*ilo*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *n*-by-*n* unitary matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine unghr interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*-1).

ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.

Application Notes

For better performance, try using *lwork* = (*ihi*-*ilo*)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of real floating-point operations is $(16/3)(ihi-ilo)^3$.

The real counterpart of this routine is orghr.

?unmhr
Multiplies an arbitrary complex matrix C by the complex unitary matrix Q determined by ?gehrd.

Syntax

```
call cunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call zunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call unmhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a matrix *C* by the unitary matrix *Q* that has been determined by a preceding call to cgehrd/zgehrd. (The routine ?gehrd reduces a real general matrix *A* to upper Hessenberg form *H* by an orthogonal similarity transformation, $A = Q*H*Q^H$, and represents the matrix *Q* as a product of *ihi-ilo elementary reflectors*. Here *ilo* and *ihi* are values determined by cgebal/zgebal when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

With ?unmhr, you can form one of the matrix products *Q***C*, Q^H**C*, *C***Q*, or *C**Q^H, overwriting the result on *C* (which may be any complex rectangular matrix). A common application of this routine is to transform a matrix *V* of eigenvectors of *H* to the matrix *QV* of eigenvectors of *A*.

Input Parameters

side	CHARACTER*1. Must be 'L' or 'R'. If *side* = 'L', then the routine forms $Q*C$ or Q^H*C. If *side* = 'R', then the routine forms $C*Q$ or $C*Q^H$.
trans	CHARACTER*1. Must be 'N' or 'C'. If *trans* = 'N', then Q is applied to C. If *trans* = 'T', then Q^H is applied to C.
m	INTEGER. The number of rows in C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
ilo, *ihi*	INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd . If $m > 0$ and *side* = 'L', then $1 \leq ilo \leq ihi \leq m$. If $m = 0$ and *side* = 'L', then *ilo* = 1 and *ihi* = 0. If $n > 0$ and *side* = 'R', then $1 \leq ilo \leq ihi \leq n$. If $n = 0$ and *side* = 'R', then *ilo* =1 and *ihi* = 0.
a, *tau*, *c*, *work*	COMPLEX for cunmhr DOUBLE COMPLEX for zunmhr. Arrays: *a*(*lda*,*) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. The second dimension of *a* must be at least max(1, *m*) if *side* = 'L' and at least max(1, *n*) if *side* = 'R'. *tau*(*) contains further details of the elementary reflectors, as returned by ?gehrd. The dimension of *tau* must be at least max (1, *m*-1) if *side* = 'L' and at least max (1, *n*-1) if *side* = 'R'. *c*(*ldc*,*) contains the *m*-by-*n* matrix C. The second dimension of *c* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*) if *side* = 'L' and at least max (1, *n*) if *side* = 'R'.
ldc	INTEGER. The leading dimension of *c*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array. If *side* = 'L', *lwork* ≥ max(1,*n*). If *side* = 'R', *lwork* ≥ max(1,*m*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c — *C* is overwritten by *Q***C*, or Q^H**C*, or *C**Q^H, or *C***Q* as specified by *side* and *trans*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine unmhr interface are the following:

a — Holds the matrix *A* of size (*r*,*r*).

r = *m* if *side* = 'L'.

r = *n* if *side* = 'R'.

tau — Holds the vector of length (*r*-1).

c — Holds the matrix *C* of size (*m*,*n*).

ilo — Default value for this argument is *ilo* = 1.

ihi — Default value for this argument is *ihi* = *n*.

side — Must be 'L' or 'R'. The default value is 'L'.

trans — Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, *lwork* should be at least *n***blocksize* if *side* = 'L' and at least *m***blocksize* if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $||E||_2 = O(\varepsilon)*||C||_2$, where ε is the machine precision.

The approximate number of floating-point operations is

$8n(ihi-ilo)^2$ if *side* = 'L';

$8m(ihi-ilo)^2$ if *side* = 'R'.

The real counterpart of this routine is ormhr.

?gebal
Balances a general matrix to improve the accuracy of computed eigenvalues and eigenvectors.

Syntax

```
call sgebal(job, n, a, lda, ilo, ihi, scale, info)
call dgebal(job, n, a, lda, ilo, ihi, scale, info)
call cgebal(job, n, a, lda, ilo, ihi, scale, info)
call zgebal(job, n, a, lda, ilo, ihi, scale, info)
call gebal(a [, scale] [,ilo] [,ihi] [,job] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine *balances* a matrix *A* by performing either or both of the following two similarity transformations:

(1) The routine first attempts to permute *A* to block upper triangular form:

$$PAP^T = A' = \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix}$$

where *P* is a permutation matrix, and A'_{11} and A'_{33} are upper triangular. The diagonal elements of A'_{11} and A'_{33} are eigenvalues of *A*. The rest of the eigenvalues of *A* are the eigenvalues of the central diagonal block A'_{22}, in rows and columns *ilo* to *ihi*. Subsequent operations to compute the eigenvalues of *A* (or its Schur factorization) need only be applied to these rows and columns; this can save a significant amount of work if $ilo > 1$ and $ihi < n$.

If no suitable permutation exists (as is often the case), the routine sets $ilo = 1$ and $ihi = n$, and A'_{22} is the whole of *A*.

(2) The routine applies a diagonal similarity transformation to A', to make the rows and columns of A'_{22} as close in norm as possible:

$$A'' = DA'D^{-1} = \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & I \end{bmatrix} \times \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix} \times \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22}^{-1} & 0 \\ 0 & 0 & I \end{bmatrix}$$

This scaling can reduce the norm of the matrix (that is, $||A''_{22}|| < ||A'_{22}||$), and hence reduce the effect of rounding errors on the accuracy of computed eigenvalues and eigenvectors.

Input Parameters

job

`CHARACTER*1`. Must be `'N'` or `'P'` or `'S'` or `'B'`.

If *job* = `'N'`, then *A* is neither permuted nor scaled (but *ilo*, *ihi*, and *scale* get their values).

If *job* = `'P'`, then *A* is permuted but not scaled.

If *job* = `'S'`, then *A* is scaled but not permuted.

If *job* = `'B'`, then *A* is both scaled and permuted.

n

`INTEGER`. The order of the matrix *A* ($n \geq 0$).

a

`REAL` for `sgebal`

`DOUBLE PRECISION` for `dgebal`

`COMPLEX` for `cgebal`

`DOUBLE COMPLEX` for `zgebal`.

Array *a*(*lda*,*) contains the matrix *A*.

The second dimension of *a* must be at least max(1, *n*). *a* is not referenced if *job* = `'N'`.

lda

`INTEGER`. The leading dimension of *a*; at least max(1, *n*).

Output Parameters

a

Overwritten by the balanced matrix (*a* is not referenced if *job* = `'N'`).

ilo, *ihi*

`INTEGER`. The values *ilo* and *ihi* such that on exit *a*(*i*,*j*) is zero if $i > j$ and $1 \leq j < ilo$ or $ihi < j \leq n$.

If *job* = `'N'` or `'S'`, then *ilo* = 1 and *ihi* = *n*.

scale

`REAL` for single-precision flavors `DOUBLE PRECISION` for double-precision flavors

Array, size at least max(1, *n*).

Contains details of the permutations and scaling factors.

More precisely, if p_j is the index of the row and column interchanged with row and column *j*, and d_j is the scaling factor used to balance row and column *j*, then

scale(*j*) = p_j for *j* = 1, 2,..., *ilo*-1, *ihi*+1,..., *n*;

	scale(*j*) = d_j for *j* = *ilo*, *ilo* + 1,..., *ihi*. The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gebal interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
scale	Holds the vector of length *n*.
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
job	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

Application Notes

The errors are negligible, compared with those in subsequent computations.

If the matrix *A* is balanced by this routine, then any eigenvectors computed subsequently are eigenvectors of the matrix *A*'' and hence you must call gebak to transform them back to eigenvectors of *A*.

If the Schur vectors of *A* are required, do not call this routine with *job* = 'S' or 'B', because then the balancing transformation is not orthogonal (not unitary for complex flavors).

If you call this routine with *job* = 'P', then any Schur vectors computed subsequently are Schur vectors of the matrix *A*'', and you need to call gebak (with *side* = 'R') to transform them back to Schur vectors of *A*.

The total number of floating-point operations is proportional to n^2.

?gebak

Transforms eigenvectors of a balanced matrix to those of the original nonsymmetric matrix.

Syntax

```
call sgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call dgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call cgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call zgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call gebak(v, scale [,ilo] [,ihi] [,job] [,side] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine is intended to be used after a matrix *A* has been balanced by a call to ?gebal, and eigenvectors of the balanced matrix A''_{22} have subsequently been computed. For a description of balancing, see gebal. The balanced matrix *A*'' is obtained as $A'' = D*P*A*P^T*\mathrm{inv}(D)$, where *P* is a permutation matrix and *D* is a diagonal scaling matrix. This routine transforms the eigenvectors as follows:

if *x* is a right eigenvector of *A*'', then $P^T*\mathrm{inv}(D)*x$ is a right eigenvector of *A*; if *y* is a left eigenvector of *A*'', then P^T*D*y is a left eigenvector of *A*.

Input Parameters

job — CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'. The same parameter *job* as supplied to ?gebal.

side — CHARACTER*1. Must be 'L' or 'R'.

If *side* = 'L', then left eigenvectors are transformed.

If *side* = 'R', then right eigenvectors are transformed.

n — INTEGER. The number of rows of the matrix of eigenvectors ($n \geq 0$).

ilo, *ihi* — INTEGER. The values *ilo* and *ihi*, as returned by ?gebal. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;

if $n = 0$, then $ilo = 1$ and $ihi = 0$.)

scale — REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors

Array, size at least max(1, *n*).

Contains details of the permutations and/or the scaling factors used to balance the original general matrix, as returned by ?gebal.

m — INTEGER. The number of columns of the matrix of eigenvectors ($m \geq 0$).

v — REAL for sgebak

DOUBLE PRECISION for dgebak

COMPLEX for cgebak

DOUBLE COMPLEX for zgebak.

Arrays:

v(*ldv*,*) contains the matrix of left or right eigenvectors to be transformed.

The second dimension of *v* must be at least max(1, *m*).

ldv — INTEGER. The leading dimension of *v*; at least max(1, *n*) .

Output Parameters

v — Overwritten by the transformed eigenvectors.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gebak` interface are the following:

v	Holds the matrix *V* of size (*n*,*m*).
scale	Holds the vector of length *n*.
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
job	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.
side	Must be 'L' or 'R'. The default value is 'L'.

Application Notes

The errors in this routine are negligible.

The approximate number of floating-point operations is approximately proportional to *m***n*.

?hseqr
Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

```
call shseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call dhseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call chseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
call zhseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
call hseqr(h, wr, wi [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
call hseqr(h, w [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally the Schur factorization, of an upper Hessenberg matrix *H*: $H = Z*T*Z^H$, where *T* is an upper triangular (or, for real flavors, quasi-triangular) matrix (the Schur form of *H*), and *Z* is the unitary or orthogonal matrix whose columns are the Schur vectors z_i.

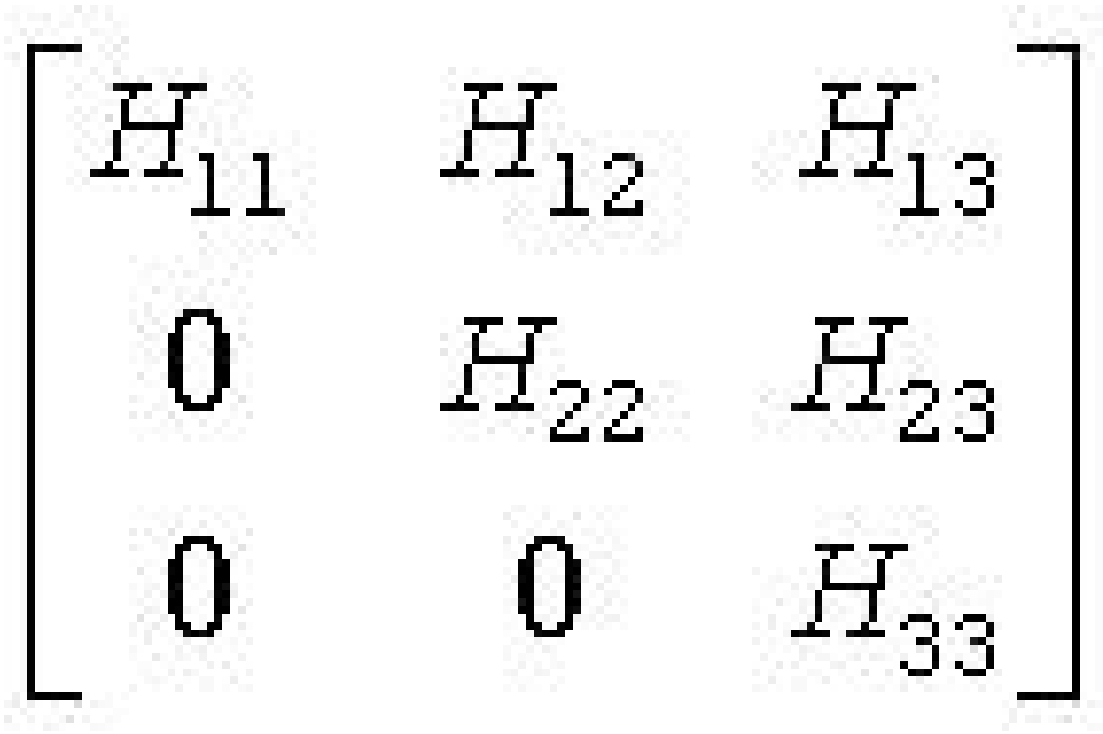
You can also use this routine to compute the Schur factorization of a general matrix *A* which has been reduced to upper Hessenberg form *H*:

$A = Q*H*Q^H$, where *Q* is unitary (orthogonal for real flavors);

$A = (QZ)*T*(QZ)^H$.

In this case, after reducing *A* to Hessenberg form by gehrd, call orghr to form *Q* explicitly and then pass *Q* to ?hseqr with *compz* = 'V'.

You can also call gebal to balance the original matrix before reducing it to Hessenberg form by ?hseqr, so that the Hessenberg matrix *H* will have the structure:



where H_{11} and H_{33} are upper triangular.

If so, only the central diagonal block H_{22} (in rows and columns *ilo* to *ihi*) needs to be further reduced to Schur form (the blocks H_{12} and H_{23} are also affected). Therefore the values of *ilo* and *ihi* can be supplied to ?hseqr directly. Also, after calling this routine you must call gebak to permute the Schur vectors of the balanced matrix to those of the original matrix.

If ?gebal has not been called, however, then *ilo* must be set to 1 and *ihi* to *n*. Note that if the Schur factorization of *A* is required, ?gebal must not be called with *job* = 'S' or 'B', because the balancing transformation is not unitary (for real flavors, it is not orthogonal).

?hseqr uses a multishift form of the upper Hessenberg *QR* algorithm. The Schur vectors are normalized so that $||z_i||_2 = 1$, but are determined only to within a complex factor of absolute value 1 (for the real flavors, to within a factor ±1).

Input Parameters

job

CHARACTER*1. Must be 'E' or 'S'.

If *job* = 'E', then eigenvalues only are required.

If *job* = 'S', then the Schur form *T* is required.

compz

CHARACTER*1. Must be 'N' or 'I' or 'V'.

If *compz* = 'N', then no Schur vectors are computed (and the array *z* is not referenced).

If *compz* = 'I', then the Schur vectors of *H* are computed (and the array *z* is initialized by the routine).

If *compz* = 'V', then the Schur vectors of *A* are computed (and the array *z* must contain the matrix *Q* on entry).

n

INTEGER. The order of the matrix *H* (*n* ≥ 0).

ilo, *ihi*

INTEGER. If *A* has been balanced by ?gebal, then *ilo* and *ihi* must contain the values returned by ?gebal. Otherwise, *ilo* must be set to 1 and *ihi* to *n*.

h, *z*, *work*

REAL for shseqr

DOUBLE PRECISION for dhseqr

COMPLEX for chseqr

DOUBLE COMPLEX for zhseqr.

Arrays:

h(*ldh*,*)) The *n*-by-*n* upper Hessenberg matrix *H*.

The second dimension of *h* must be at least max(1, *n*).

z(*ldz*,*)

If *compz* = 'V', then *z* must contain the matrix *Q* from the reduction to Hessenberg form.

If *compz* = 'I', then *z* need not be set.

If *compz* = 'N', then *z* is not referenced.

The second dimension of *z* must be

at least max(1, *n*) if *compz* = 'V' or 'I';

at least 1 if *compz* = 'N'.

work(*lwork*) is a workspace array.

The dimension of *work* must be at least max (1, *n*).

ldh

INTEGER. The leading dimension of *h*; at least max(1, *n*).

ldz

INTEGER. The leading dimension of *z*;

If *compz* = 'N', then *ldz* ≥ 1.

If *compz* = 'V' or 'I', then *ldz* ≥ max(1, *n*).

lwork

INTEGER. The dimension of the array *work*.

lwork ≥ max(1, *n*) is sufficient and delivers very good and sometimes optimal performance. However, *lwork* as large as 11**n* may be required for optimal performance. A workspace query is recommended to determine the optimal workspace size.

If *lwork* = -1, then a workspace query is assumed; the routine only estimates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

w

COMPLEX for chseqr

DOUBLE COMPLEX for zhseqr.

Array, size at least max (1, *n*). Contains the computed eigenvalues, unless *info*>0. The eigenvalues are stored in the same order as on the diagonal of the Schur form *T* (if computed).

wr, *wi*

`REAL` for `shseqr`

`DOUBLE PRECISION` for `dhseqr`

Arrays, size at least max (1, *n*) each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues, unless *info* > 0. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first. The eigenvalues are stored in the same order as on the diagonal of the Schur form *T* (if computed).

h

If *info* = 0 and *job* = 'S', *h* contains the upper quasi-triangular matrix *T* from the Schur decomposition (the Schur form).

If *info* = 0 and *job* = 'E', the contents of *h* are unspecified on exit. (The output value of *h* when *info* > 0 is given under the description of *info* below.)

z

If *compz* = 'V' and *info* = 0, then *z* contains *Q*Z*.

If *compz* = 'I' and *info* = 0, then *z* contains the unitary or orthogonal matrix *Z* of the Schur vectors of *H*.

If *compz* = 'N', then *z* is not referenced.

work(1)

On exit, if *info* = 0, then *work(1)* returns the optimal *lwork*.

info

`INTEGER`.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, `?hseqr` failed to compute all of the eigenvalues. Elements 1,2, ..., *ilo*-1 and *i*+1, *i*+2, ..., *n* of the eigenvalue arrays (*wr* and *wi* for real flavors and *w* for complex flavors) contain the real and imaginary parts of those eigenvalues that have been successfully found.

If *info* > 0, and *job* = 'E', then on exit, the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *H*.

If *info* > 0, and *job* = 'S', then on exit (initial value of *H*)**U* = *U**(final value of *H*), where *U* is a unitary matrix. The final value of *H* is upper Hessenberg and triangular in rows and columns *info*+1 through *ihi*.

If *info* > 0, and *compz* = 'V', then on exit (final value of *Z*) = (initial value of *Z*)**U*, where *U* is the unitary matrix (regardless of the value of *job*).

If *info* > 0, and *compz* = 'I', then on exit (final value of *Z*) = *U*, where *U* is the unitary matrix (regardless of the value of *job*).

If *info* > 0, and *compz* = 'N', then *Z* is not accessed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hseqr` interface are the following:

h	Holds the matrix *H* of size (*n*,*n*).
wr	Holds the vector of length *n*. Used in real flavors only.
wi	Holds the vector of length *n*. Used in real flavors only.
w	Holds the vector of length *n*. Used in complex flavors only.
z	Holds the matrix *Z* of size (*n*,*n*).
job	Must be `'E'` or `'S'`. The default value is `'E'`.
compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = `'I'`, if *z* is present, *compz* = `'N'`, if *z* is omitted. If present, *compz* must be equal to `'I'` or `'V'` and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted.

Application Notes

The computed Schur factorization is the exact factorization of a nearby matrix $H + E$, where $||E||_2 < O(\varepsilon)||H||_2/s_i$, and ε is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then $|\lambda_i - \mu_i| \le c(n)*\varepsilon*||H||_2/s_i$, where $c(n)$ is a modestly increasing function of n, and s_i is the reciprocal condition number of λ_i. The condition numbers s_i may be computed by calling trsna.

The total number of floating-point operations depends on how rapidly the algorithm converges; typical numbers are as follows.

If only eigenvalues are computed:	$7n^3$ for real flavors $25n^3$ for complex flavors.
If the Schur form is computed:	$10n^3$ for real flavors $35n^3$ for complex flavors.
If the full Schur factorization is computed:	$20n^3$ for real flavors $70n^3$ for complex flavors.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hsein
Computes selected eigenvectors of an upper Hessenberg matrix that correspond to specified eigenvalues.

Syntax

```
call shsein(side, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr, mm, m, work, ifaill, ifailr, info)
call dhsein(side, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr, mm, m, work, ifaill, ifailr, info)
call chsein(side, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm, m, work, rwork, ifaill, ifailr, info)
call zhsein(side, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm, m, work, rwork, ifaill, ifailr, info)
call hsein(h, wr, wi, select [, vl] [,vr] [,ifaill] [,ifailr] [,initv] [,eigsrc] [,m] [,info])
call hsein(h, w, select [,vl] [,vr] [,ifaill] [,ifailr] [,initv] [,eigsrc] [,m] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes left and/or right eigenvectors of an upper Hessenberg matrix *H*, corresponding to selected eigenvalues.

The right eigenvector *x* and the left eigenvector *y*, corresponding to an eigenvalue λ, are defined by: $H*x = \lambda*x$ and $y^H*H = \lambda*y^H$ (or $H^H*y = \lambda^**y$). Here λ^* denotes the conjugate of λ.

The eigenvectors are computed by inverse iteration. They are scaled so that, for a real eigenvector x, $\max|x_i| = 1$, and for a complex eigenvector, $\max(|\mathrm{Re}x_i| + |\mathrm{Im}x_i|) = 1$.

If *H* has been formed by reduction of a general matrix *A* to upper Hessenberg form, then eigenvectors of *H* may be transformed to eigenvectors of *A* by ormhr or unmhr.

Input Parameters

side

CHARACTER*1. Must be 'R' or 'L' or 'B'.

If *side* = 'R', then only right eigenvectors are computed.

If *side* = 'L', then only left eigenvectors are computed.

If *side* = 'B', then all eigenvectors are computed.

eigsrc

CHARACTER*1. Must be 'Q' or 'N'.

If *eigsrc* = 'Q', then the eigenvalues of *H* were found using hseqr; thus if *H* has any zero sub-diagonal elements (and so is block triangular), then the *j*-th eigenvalue can be assumed to be an eigenvalue of the block containing

the *j*-th row/column. This property allows the routine to perform inverse iteration on just one diagonal block. If *eigsrc* = 'N', then no such assumption is made and the routine performs inverse iteration using the whole matrix.

initv — CHARACTER*1. Must be 'N' or 'U'.

If *initv* = 'N', then no initial estimates for the selected eigenvectors are supplied.

If *initv* = 'U', then initial estimates for the selected eigenvectors are supplied in *vl* and/or *vr*.

select — LOGICAL.

Array, size at least max (1, *n*). Specifies which eigenvectors are to be computed.

For real flavors:

To obtain the real eigenvector corresponding to the real eigenvalue *wr*(*j*), set *select*(*j*) to .TRUE.

To select the complex eigenvector corresponding to the complex eigenvalue (*wr*(*j*), *wi*(*j*)) with complex conjugate (*wr*(*j*+1), *wi*(*j*+1)), set *select*(*j*) and/or *select*(*j*+1) to .TRUE.; the eigenvector corresponding to the first eigenvalue in the pair is computed.

For complex flavors:

To select the eigenvector corresponding to the eigenvalue *w*(*j*), set *select*(*j*) to .TRUE.

n — INTEGER. The order of the matrix *H* (*n*≥ 0).

h, *vl*, *vr*, *work* — REAL for shsein

DOUBLE PRECISION for dhsein

COMPLEX for chsein

DOUBLE COMPLEX for zhsein.

Arrays:

h(*ldh*,*) The *n*-by-*n* upper Hessenberg matrix *H*. If an NAN value is detected in *h*, the routine returns with *info* = -6.

The second dimension of *h* must be at least max(1, *n*).

vl(*ldvl*,*)

If *initv* = 'V' and *side* = 'L' or 'B', then *vl* must contain starting vectors for inverse iteration for the left eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.

If *initv* = 'N', then *vl* need not be set.

The second dimension of *vl* must be at least max(1, *mm*) if *side* = 'L' or 'B' and at least 1 if *side* = 'R'.

The array *vl* is not referenced if *side* = 'R'.

vr(*ldvr*,*)

If *initv* = 'V' and *side* = 'R' or 'B', then *vr* must contain starting vectors for inverse iteration for the right eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.

If *initv* = 'N', then *vr* need not be set.

The second dimension of *vr* must be at least max(1, *mm*) if *side* = 'R' or 'B' and at least 1 if *side* = 'L'.

The array *vr* is not referenced if *side* = 'L'.

work(*) is a workspace array.

size at least max (1, *n**(*n*+*2*)) for real flavors and at least max (1, *n***n*) for complex flavors.

ldh INTEGER. The leading dimension of *h*; at least max(1, *n*).

w COMPLEX for chsein

DOUBLE COMPLEX for zhsein.

Array, size at least max (1, *n*).

Contains the eigenvalues of the matrix *H*.

If *eigsrc* = 'Q', the array must be exactly as returned by ?hseqr.

wr, *wi* REAL for shsein

DOUBLE PRECISION for dhsein

Arrays, size at least max (1, *n*) each.

Contain the real and imaginary parts, respectively, of the eigenvalues of the matrix *H*. Complex conjugate pairs of values must be stored in consecutive elements of the arrays. If *eigsrc* = 'Q', the arrays must be exactly as returned by ?hseqr.

ldvl INTEGER. The leading dimension of *vl*.

If *side* = 'L' or 'B', *ldvl*≥ max(1,*n*) .

If *side* = 'R', *ldvl*≥ 1.

ldvr INTEGER. The leading dimension of *vr*.

If *side* = 'R' or 'B', *ldvr*≥ max(1,*n*) .

If *side* = 'L', *ldvr*≥1.

mm INTEGER. The number of columns in *vl* and/or *vr*.

Must be at least *m*, the actual number of columns required (see *Output Parameters* below).

For real flavors, *m* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector (see *select*).

For complex flavors, *m* is the number of selected eigenvectors (see *select*).

Constraint:

0 ≤*mm*≤*n*.

rwork

REAL for chsein

DOUBLE PRECISION for zhsein.

Array, size at least max (1, *n*).

Output Parameters

select

Overwritten for real flavors only.

If a complex eigenvector was selected as specified above, then *select*(*j*) is set to .TRUE. and *select*(*j* + 1) to .FALSE.

w

The real parts of some elements of *w* may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

wr

Some elements of *wr* may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

vl, *vr*

If *side* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *select*).

If *side* = 'R' or 'B', *vr* contains the computed right eigenvectors (as specified by *select*).

The eigenvectors treated column-wise form a rectangular *n*-by-*mm* matrix.

For real flavors: a real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns: the first column holds the real part of the eigenvector and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by *matrix_layout* (using either column major or row major layout).

m

INTEGER. *For real flavors*: the number of columns of *vl* and/or *vr* required to store the selected eigenvectors.

For complex flavors: the number of selected eigenvectors.

ifaill, *ifailr*

INTEGER.

Arrays, size at least max(1, *mm*) each.

ifaill(*i*) = 0 if the *i*th column of *vl* converged;

ifaill(*i*) = *j* > 0 if the eigenvector stored in the *i*-th column of *vl* (corresponding to the *j*th eigenvalue) failed to converge.

ifailr(*i*) = 0 if the *i*th column of *vr* converged;

ifailr(*i*) = *j* > 0 if the eigenvector stored in the *i*-th column of *vr* (corresponding to the *j*th eigenvalue) failed to converge.

For real flavors: if the *i*th and (*i*+1)th columns of *vl* contain a selected complex eigenvector, then *ifaill*(*i*) and *ifaill*(*i* + 1) are set to the same value. A similar rule holds for *vr* and *ifailr*.

The array *ifaill* is not referenced if *side* = 'R'. The array *ifailr* is not referenced if *side* = 'L'.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, then *i* eigenvectors (as indicated by the parameters *ifaill* and/or *ifailr* above) failed to converge. The corresponding columns of *vl* and/or *vr* contain no useful information.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hsein` interface are the following:

h Holds the matrix *H* of size (*n*,*n*).

wr Holds the vector of length *n*. Used in real flavors only.

wi Holds the vector of length *n*. Used in real flavors only.

w Holds the vector of length *n*. Used in complex flavors only.

select Holds the vector of length *n*.

vl Holds the matrix *VL* of size (*n*,*mm*).

vr Holds the matrix *VR* of size (*n*,*mm*).

ifaill Holds the vector of length (*mm*). Note that there will be an error condition if *ifaill* is present and *vl* is omitted.

ifailr Holds the vector of length (*mm*). Note that there will be an error condition if *ifailr* is present and *vr* is omitted.

initv Must be `'N'` or `'U'`. The default value is `'N'`.

eigsrc Must be `'N'` or `'Q'`. The default value is `'N'`.

side Restored based on the presence of arguments *vl* and *vr* as follows:

side = `'B'`, if both *vl* and *vr* are present,

side = `'L'`, if *vl* is present and *vr* omitted,

side = `'R'`, if *vl* is omitted and *vr* present,

Note that there will be an error condition if both *vl* and *vr* are omitted.

Application Notes

Each computed right eigenvector *x* i is the exact eigenvector of a nearby matrix $A + E_i$, such that $||E_i|| < O(\varepsilon)||A||$. Hence the residual is small:

$||Ax_i - \lambda_i x_i|| = O(\varepsilon)||A||$.

However, eigenvectors corresponding to close or coincident eigenvalues may not accurately span the relevant subspaces.

Similar remarks apply to computed left eigenvectors.

?trevc

Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr.

Syntax

```
call strevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, info)
call dtrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, info)
call ctrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
call ztrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
call trevc(t [, howmny] [,select] [,vl] [,vr] [,m] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes some or all of the right and/or left eigenvectors of an upper triangular matrix *T* (or, for real flavors, an upper quasi-triangular matrix *T*). Matrices of this type are produced by the Schur factorization of a general matrix: $A = Q*T*Q^H$, as computed by hseqr.

The right eigenvector *x* and the left eigenvector *y* of *T* corresponding to an eigenvalue *w*, are defined by:

$T*x = w*x$, $y^H*T = w*y^H$, where y^H denotes the conjugate transpose of *y*.

The eigenvalues are not input to this routine, but are read directly from the diagonal blocks of *T*.

This routine returns the matrices *X* and/or *Y* of right and left eigenvectors of *T*, or the products *Q***X* and/or *Q***Y*, where *Q* is an input matrix.

If *Q* is the orthogonal/unitary factor that reduces a matrix *A* to Schur form *T*, then *Q***X* and *Q***Y* are the matrices of right and left eigenvectors of *A*.

Input Parameters

side

CHARACTER*1. Must be 'R' or 'L' or 'B'.

If *side* = 'R', then only right eigenvectors are computed.

If *side* = 'L', then only left eigenvectors are computed.

If *side* = 'B', then all eigenvectors are computed.

howmny

CHARACTER*1. Must be 'A' or 'B' or 'S'.

If *howmny* = 'A', then all eigenvectors (as specified by *side*) are computed.

If *howmny* = 'B', then all eigenvectors (as specified by *side*) are computed and backtransformed by the matrices supplied in *vl* and *vr*.

If *howmny* = 'S', then selected eigenvectors (as specified by *side* and *select*) are computed.

select

LOGICAL.

Array, size at least max (1, *n*).

If *howmny* = 'S', *select* specifies which eigenvectors are to be computed.

If *howmny* = 'A' or 'B', *select* is not referenced.

For real flavors:

If *omega*(*j*) is a real eigenvalue, the corresponding real eigenvector is computed if *select*(*j*) is .TRUE..

If *omega*(*j*) and *omega*(*j* + 1) are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either *select*(*j*) or *select*(*j* + 1) is .TRUE., and on exit *select*(*j*) is set to .TRUE.and *select*(*j* + 1) is set to .FALSE..

For complex flavors:

The eigenvector corresponding to the *j*-th eigenvalue is computed if *select*(*j*) is .TRUE..

n — INTEGER. The order of the matrix *T* (*n*≥ 0).

t, *vl*, *vr* — REAL for strevc

DOUBLE PRECISION for dtrevc

COMPLEX for ctrevc

DOUBLE COMPLEX for ztrevc.

Arrays:

t(*ldt*,*) contains the *n*-by-*n* matrix *T* in Schur canonical form. For complex flavors ctrevc and ztrevc, contains the upper triangular matrix *T*.

The second dimension of *t* must be at least max(1, *n*).

vl(*ldvl*,*)

If *howmny* = 'B' and *side* = 'L' or 'B', then *vl* must contain an *n*-by-*n* matrix *Q* (usually the matrix of Schur vectors returned by ?hseqr).

If *howmny* = 'A' or 'S', then *vl* need not be set.

The second dimension of *vl* must be at least max(1, *mm*) if *side* = 'L' or 'B' and at least 1 if *side* = 'R'.

The array *vl* is not referenced if *side* = 'R'.

vr(*ldvr*,*)

If *howmny* = 'B' and *side* = 'R' or 'B', then *vr* must contain an *n*-by-*n* matrix *Q* (usually the matrix of Schur vectors returned by ?hseqr). .

If *howmny* = 'A' or 'S', then *vr* need not be set.

The second dimension of *vr* must be at least max(1, *mm*) if *side* = 'R' or 'B' and at least 1 if *side* = 'L'.

The array *vr* is not referenced if *side* = 'L'.

work(*) is a workspace array.

size at least max (1, 3**n*) for real flavors and at least max (1, 2**n*) for complex flavors.

ldt — INTEGER. The leading dimension of *t*; at least max(1, *n*).

ldvl INTEGER. The leading dimension of *vl*.

If *side* = 'L' or 'B', *ldvl*≥*n*.

If *side* = 'R', *ldvl*≥ 1.

ldvr INTEGER. The leading dimension of *vr*.

If *side* = 'R' or 'B', *ldvr*≥*n*.

If *side* = 'L', *ldvr*≥ 1.

mm INTEGER. The number of columns in the arrays *vl* and/or *vr*. Must be at least *m* (the precise number of columns required).

If *howmny* = 'A' or 'B', *mm* = *n*.

If *howmny* = 'S': *for real flavors*, *mm* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector;

for complex flavors, *mm* is the number of selected eigenvectors (see *select*).

Constraint: 0 ≤*mm*≤*n*.

rwork REAL for ctrevc

DOUBLE PRECISION for ztrevc.

Workspace array, size at least max (1, *n*).

Output Parameters

select If a complex eigenvector of a real matrix was selected as specified above, then *select*(*j*) is set to .TRUE. and *select*(*j* + 1) to .FALSE.

t COMPLEX for ctrevc

DOUBLE COMPLEX for ztrevc.

ctrevc/ztrevc modify the *t*(*ldt*,*) array, which is restored on exit.

vl, *vr* If *side* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *howmny* and *select*).

If *side* = 'R' or 'B', *vr* contains the computed right eigenvectors (as specified by *howmny* and *select*).

The eigenvectors treated column-wise form a rectangular *n*-by-*mm* matrix.

For real flavors: a real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns: the first column holds the real part of the eigenvector and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by *matrix_layout* (using either column major or row major layout).

m INTEGER.

For complex flavors: the number of selected eigenvectors.

If *howmny* = 'A' or 'B', *m* is set to *n*.

For real flavors: the number of columns of *vl* and/or *vr* actually used to store the selected eigenvectors.

If *howmny* = 'A' or 'B', *m* is set to *n*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine trevc interface are the following:

t — Holds the matrix *T* of size (*n*,*n*).

select — Holds the vector of length *n*.

vl — Holds the matrix *VL* of size (*n*,*mm*).

vr — Holds the matrix *VR* of size (*n*,*mm*).

side — If omitted, this argument is restored based on the presence of arguments *vl* and *vr* as follows:

side = 'B', if both *vl* and *vr* are present,

side = 'L', if *vr* is omitted,

side = 'R', if *vl* is omitted.

Note that there will be an error condition if both *vl* and *vr* are omitted.

howmny — If omitted, this argument is restored based on the presence of argument *select* as follows:

howmny = 'V', if *q* is present,

howmny = 'N', if *q* is omitted.

If present, *vect* = 'V' or 'U' and the argument *q* must also be present.

Note that there will be an error condition if both *select* and *howmny* are present.

Application Notes

If x_i is an exact right eigenvector and y_i is the corresponding computed eigenvector, then the angle $\theta(y_i, x_i)$ between them is bounded as follows: $\theta(y_i, x_i) \leq (c(n)\varepsilon ||T||_2)/\text{sep}_i$ where sep_i is the reciprocal condition number of x_i. The condition number sep_i may be computed by calling ?trsna.

?trevc3
Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr using Level 3 BLAS

Syntax

```
call strevc3(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, lwork, info)
```

```
call dtrevc3(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, lwork, info)
call ctrevc3(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, lwork, rwork, lrwork, info)
call ztrevc3(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, lwork, rwork, lrwork, info)
```

Include Files

- mkl.fi

Description

This routine computes some or all of the right and left eigenvectors of an upper triangular matrix *T* (or, for real flavors, an upper quasi-triangular matrix *T*) using Level 3 BLAS. Matrices of this type are produced by the Schur factorization of a general matrix: `A =Q*T*QH`, as computed by `hseqr`.

The right eigenvector *x* and the left eigenvector *y* of *T* corresponding to an eigenvalue *w* are defined by the following:

$T*x = w*x, y^H*T = w*y^H$

where y^H denotes the conjugate transpose of *y*.

The eigenvalues are not passed to this routine but are read directly from the diagonal blocks of *T*.

This routine returns one or both of the matrices *X* and *Y* of the right and left eigenvectors of *T*, or one or both of the products `Q*X` and `Q*Y`, where *Q* is an input matrix.

If *Q* is the orthogonal/unitary factor that reduces a matrix *A* to Schur form *T*, then `Q*X` and `Q*Y` are the matrices of the right and left eigenvectors of *A*.

Input Parameters

side

`CHARACTER*1`

Must be `'R'`, `'L'`, or `'B'`.

- If *side* = `'R'`, only right eigenvectors are computed.
- If *side* = `'L'`, only left eigenvectors are computed.
- If *side* = `'B'`, all eigenvectors are computed.

howmny

`CHARACTER*1`

Must be `'A'`, `'B'`, or `'S'`.

- If *howmny* = `'A'`, all eigenvectors (as specified by *side*) are computed.
- If *howmny* = `'B'`, all eigenvectors (as specified by *side*) are computed and back-transformed by the matrices supplied in *vl* and *vr*.
- If *howmny* = `'S'`, selected eigenvectors (as specified by *side* and *select*) are computed.

select

Array with a size of at least `max (1, n)`

If *howmny* = `'S'`, *select* specifies which eigenvectors are to be computed. If *howmny* = `'A'` or *howmny* = `'B'`, *select* is not referenced.

For real flavors:

- If `omega(j)` is a real eigenvalue and `select(j)` is `.TRUE.`, the corresponding real eigenvector is computed.
- If `omega(j)` and `omega(j + 1)` are the real and imaginary parts of a complex eigenvalue and either `select(j)` or `select(j + 1)` is `.TRUE.`, the corresponding complex eigenvector is computed, and on exit `select(j)` is set to `.TRUE.` and `select(j + 1)` is set to `.FALSE.`.

For complex flavors:

- If `select(j)` is `.TRUE.`, the eigenvector corresponding to the j^{th} eigenvalue is computed.

n

`INTEGER`

The order of the matrix `T` ($n \geq 0$).

t, *vl*, *vr*, *work*

- `REAL` for `strevc3`
- `DOUBLE PRECISION` for `dtrevc3`
- `COMPLEX` for `ctrevc3`
- `DOUBLE COMPLEX` for `ztrevc3`

Arrays:

- `t(ldt,*)` contains the *n*-by-*n* matrix *T* in Schur canonical form. For complex flavors `ctrevc3` and `ztrevc3`, the array contains the upper triangular matrix *T*.

  The second dimension of *t* must be at least `max(1, n)`.
- `vl(ldvl,*)`

  If *howmny* = `'B'` and *side* = `'L'` or `'B'`, then *vl* must contain an *n*-by-*n* matrix *Q* (usually the matrix of Schur vectors returned by `?hseqr`).

  If *howmny* = `'A'` or `'S'`, *vl* need not be set.

  The second dimension of *vl* must be at least `max(1, mm)` if *side* = `'L'` or `'B'`, and at least `1` if *side* = `'R'`.

  The array *vl* is not referenced if *side* = `'R'`.
- `vr(ldvr,*)`

  If *howmny* = `'B'` and *side* = `'R'` or `'B'`, *vr* must contain an *n*-by-*n* matrix *Q* (usually the matrix of Schur vectors returned by `?hseqr`).

  If *howmny* = `'A'` or `'S'`, *vr* need not be set.

  The second dimension of *vr* must be at least `max(1, mm)` if *side* = `'R'` or `'B'`, and at least `1` if *side* = `'L'`.

  The array *vr* is not referenced if *side* = `'L'`.
- `work(*)` is a workspace array, and its dimension is `max (1, lwork)`.

lwork

`INTEGER`

The size of the work array. Must be at least `max(1, 3*n)` for real flavors, and at least `max(1, 2*n)` for complex flavors.

If *lwork* = -1, a workspace query is assumed; the routine calculates only the optimal size of the work array and returns this value as the first entry of the work array, and no error message related to *lwork* is issued by xerbla. For details, see "Application Notes" below.

ldt

INTEGER

The leading dimension of *t*. It is at least max(1, n).

ldvl

INTEGER

The leading dimension of *vl*.

- If *side* = 'L' or 'B', ldvl≥n.
- If *side* = 'R', ldvl≥ 1.

ldvr

INTEGER

The leading dimension of *vr*.

- If *side* = 'R' or 'B', ldvr≥n.
- If *side* = 'L', ldvr≥ 1.

mm

INTEGER

The number of columns in one or both of the arrays *vl* and *vr*. Must be at least *m* (the precise number of columns required).

- If *howmny* = 'A' or 'B', mm = n.
- If *howmny* = 'S': for real flavors, *mm* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector; for complex flavors, *mm* is the number of selected eigenvectors (see *select*).

Constraint: 0 ≤*mm*≤*n*.

rwork

- REAL for ctrevc3
- DOUBLE PRECISION for ztrevc3

The workspace array is used in complex flavors only. Its dimensionis max (1, lrwork).

lrwork

INTEGER

The size of the *rwork* array. It must be at least max(1, n).

If *lrwork* = -1, a workspace query is assumed; the routine calculates only the optimal size of the work array and returns this value as the first entry of the *rwork* array, and no error message related to *lrwork* is issued by xerbla. For details, see "Application Notes" below.

Output Parameters

select

If a complex eigenvector of a real matrix was selected as specified above, then select(j) is set to .TRUE. and select(j + 1) is set to .FALSE..

t

COMPLEX for ctrevc3

DOUBLE COMPLEX for ztrevc3

`ctrevc3` or `ztrevc3` modifies the `t(ldt,*)` array, which is restored on exit.

vl, *vr*

If *side* = `'L'` or `'B'`, *vl* contains the computed left eigenvectors (as specified by *howmny* and *select*).

If *side* = `'R'` or `'B'`, *vr* contains the computed right eigenvectors (as specified by *howmny* and *select*).

Treated column-wise, the eigenvectors form a rectangular *n*-by-*mm* matrix.

For real flavors A real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns. The first column holds the real part of the eigenvector, and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by `matrix_layout` (using either column major or row major layout).

m

`INTEGER`

For complex flavors The number of selected eigenvectors. If *howmny* = `'A'` or `'B'`, *m* is set to *n*.

For real flavors The number of columns of one or both of *vl* and *vr* actually used to store the selected eigenvectors. If *howmny* = `'A'` or `'B'`, *m* is set to *n*.

work(1)

On exit, if *info* = `0`, *work(1)* returns the required optimal size of *lwork*.

rwork(1)

On exit, if *info* = `0`, then *rwork(1)* returns the required optimal size of *lrwork*.

info

`INTEGER`

If *info* = `0`, the execution is successful.

If *info* = `-i`, the i^{th} parameter contained an illegal value.

Application Notes

If *xi* is an exact right eigenvector and *yi* is the corresponding computed eigenvector, the angle `θ(yi, xi)` between them is bounded as follows:

```
θ(yi,xi)≤(c(n)ε||T||2)/sepi
```

where *sepi* is the reciprocal condition number of *xi*. You can compute the condition number *sepi* by calling `?trsna`.

See Also

Matrix Storage Schemes

?trsna

Estimates condition numbers for specified eigenvalues and right eigenvectors of an upper (quasi-) triangular matrix.

Syntax

```
call strsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work, ldwork, iwork, info)
call dtrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work, ldwork, iwork, info)
call ctrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work, ldwork, rwork, info)
call ztrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work, ldwork, rwork, info)
call trsna(t [, s] [,sep] [,vl] [,vr] [,select] [,m] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates condition numbers for specified eigenvalues and/or right eigenvectors of an upper triangular matrix T (or, for real flavors, upper quasi-triangular matrix T in canonical Schur form). These are the same as the condition numbers of the eigenvalues and right eigenvectors of an original matrix $A = Z*T*Z^H$ (with unitary or, for real flavors, orthogonal Z), from which T may have been derived.

The routine computes the reciprocal of the condition number of an eigenvalue λ_i as $s_i = |v^T*u|/(||u||_E||v||_E)$ for real flavors and $s_i = |v^H*u|/(||u||_E||v||_E)$ for complex flavors,

where:

- u and v are the right and left eigenvectors of T, respectively, corresponding to λ_i.
- v^T/v^H denote transpose/conjugate transpose of v, respectively.

This reciprocal condition number always lies between zero (ill-conditioned) and one (well-conditioned).

An approximate error estimate for a computed eigenvalue λ_i is then given by $\varepsilon*||T||/s_i$, where ε is the *machine precision*.

To estimate the reciprocal of the condition number of the right eigenvector corresponding to λ_i, the routine first calls trexc to reorder the diagonal elements of matrix T so that λ_i is in the leading position:

$$T = Q \begin{bmatrix} \lambda_i & C^H \\ 0 & T_{22} \end{bmatrix} Q^H$$

The reciprocal condition number of the eigenvector is then estimated as sep_i, the smallest singular value of the matrix T_{22} - λ_i*I.

An approximate error estimate for a computed right eigenvector u corresponding to λ_i is then given by ε*||T||/sep_i.

Input Parameters

job

CHARACTER*1. Must be 'E' or 'V' or 'B'.

If *job* = 'E', then condition numbers for eigenvalues only are computed.

If *job* = 'V', then condition numbers for eigenvectors only are computed.

If *job* = 'B', then condition numbers for both eigenvalues and eigenvectors are computed.

howmny

CHARACTER*1. Must be 'A' or 'S'.

If *howmny* = 'A', then the condition numbers for all eigenpairs are computed.

If *howmny* = 'S', then condition numbers for selected eigenpairs (as specified by *select*) are computed.

select

LOGICAL.

Array, size at least max (1, *n*) if *howmny* = 'S' and at least 1 otherwise.

Specifies the eigenpairs for which condition numbers are to be computed if *howmny*= 'S'.

For real flavors:

To select condition numbers for the eigenpair corresponding to the real eigenvalue λ_j, *select*(*j*) must be set .TRUE.;

to select condition numbers for the eigenpair corresponding to a complex conjugate pair of eigenvalues λ_j and λ_{j+1}), *select*(*j*) and/or *select*(*j* + 1) must be set .TRUE.

For complex flavors

To select condition numbers for the eigenpair corresponding to the eigenvalue λ_j, *select*(*j*) must be set .TRUE.*select* is not referenced if *howmny* = 'A'.

n

INTEGER. The order of the matrix *T* (*n*≥ 0).

t, *vl*, *vr*, *work*

REAL for strsna

DOUBLE PRECISION for dtrsna

COMPLEX for ctrsna

DOUBLE COMPLEX for ztrsna.

Arrays:

t(*ldt*,*) contains the *n*-by-*n* matrix *T*.

The second dimension of *t* must be at least max(1, *n*).

vl(*ldvl*,*)

If *job* = 'E' or 'B', then *vl* must contain the left eigenvectors of *T* (or of any matrix $Q*T*Q^H$ with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by trevc or hsein.

The second dimension of *vl* must be at least max(1, *mm*) if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.

The array *vl* is not referenced if *job* = 'V'.

vr(*ldvr*,*)

If *job* = 'E' or 'B', then *vr* must contain the right eigenvectors of *T* (or of any matrix $Q*T*Q^H$ with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by trevc or hsein.

The second dimension of *vr* must be at least max(1, *mm*) if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.

The array *vr* is not referenced if *job* = 'V'.

work is a workspace array, its dimension (*ldwork*,*n*+6).

The array *work* is not referenced if *job* = 'E'.

ldt — INTEGER. The leading dimension of *t*; at least max(1, *n*).

ldvl — INTEGER. The leading dimension of *vl*.

If *job* = 'E' or 'B', *ldvl* ≥ max(1,*n*) .

If *job* = 'V', *ldvl* ≥ 1.

ldvr — INTEGER. The leading dimension of *vr*.

If *job* = 'E' or 'B', *ldvr* ≥ max(1,*n*) .

If *job* = 'R', *ldvr* ≥ 1.

mm — INTEGER. The number of elements in the arrays *s* and *sep*, and the number of columns in *vl* and *vr* (if used). Must be at least *m* (the precise number required).

If *howmny* = 'A', *mm* = *n*;

if *howmny* = 'S', *for real flavorsmm* is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues.

for complex flavorsmm is the number of selected eigenpairs (see *select*). Constraint:

0 ≤*mm*≤*n*.

ldwork — INTEGER. The leading dimension of *work*.

If *job* = 'V' or 'B', *ldwork* ≥ max(1,*n*).

If *job* = 'E', *ldwork* ≥ 1.

rwork — REAL for ctrsna, ztrsna.

Array, size at least max (1, *n*). The array is not referenced if *job* = 'E'.

iwork INTEGER for strsna, dtrsna.

Array, size at least max (1, 2*(*n* - 1)). The array is not referenced if *job* = 'E'.

Output Parameters

s REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size at least max(1, *mm*) if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.

Contains the reciprocal condition numbers of the selected eigenvalues if *job* = 'E' or 'B', stored in consecutive elements of the array. Thus *s*(*j*), *sep*(*j*) and the *j*-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the *j* th eigenpair unless all eigenpairs have been selected).

For real flavors: for a complex conjugate pair of eigenvalues, two consecutive elements of *s* are set to the same value. The array *s* is not referenced if *job* = 'V'.

sep REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size at least max(1, *mm*) if *job* = 'V' or 'B' and at least 1 if *job* = 'E'. Contains the estimated reciprocal condition numbers of the selected right eigenvectors if *job* = 'V' or 'B', stored in consecutive elements of the array.

For real flavors: for a complex eigenvector, two consecutive elements of *sep* are set to the same value; if the eigenvalues cannot be reordered to compute *sep*(*j*), then *sep*(*j*) is set to zero; this can only occur when the true value would be very small anyway. The array *sep* is not referenced if *job* = 'E'.

m INTEGER.

For complex flavors: the number of selected eigenpairs.

If *howmny* = 'A', *m* is set to *n*.

For real flavors: the number of elements of *s* and/or *sep* actually used to store the estimated condition numbers.

If *howmny* = 'A', *m* is set to *n*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine trsna interface are the following:

t	Holds the matrix *T* of size (*n*,*n*).
s	Holds the vector of length (*mm*).
sep	Holds the vector of length (*mm*).
vl	Holds the matrix *VL* of size (*n*,*mm*).
vr	Holds the matrix *VR* of size (*n*,*mm*).
select	Holds the vector of length *n*.
job	Restored based on the presence of arguments *s* and sep as follows: *job* = 'B', if both *s* and *sep* are present, *job* = 'E', if *s* is present and *sep* omitted, *job* = 'V', if *s* is omitted and *sep* present. Note an error condition if both *s* and *sep* are omitted.
howmny	Restored based on the presence of the argument *select* as follows: *howmny* = 'S', if *select* is present, *howmny* = 'A', if *select* is omitted.

Note that the arguments *s*, *vl*, and *vr* must either be all present or all omitted.

Otherwise, an error condition is observed.

Application Notes

The computed values sep_i may overestimate the true value, but seldom by a factor of more than 3.

?trexc
Reorders the Schur factorization of a general matrix.

Syntax

```
call strexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call dtrexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call ctrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
call ztrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
call trexc(t, ifst, ilst [,q] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reorders the Schur factorization of a general matrix $A = Q*T*Q^H$, so that the diagonal element or block of *T* with row index *ifst* is moved to row *ilst*.

The reordered Schur form *S* is computed by an unitary (or, for real flavors, orthogonal) similarity transformation: $S = Z^H*T*Z$. Optionally the updated matrix *P* of Schur vectors is computed as $P = Q*Z$, giving $A = P*S*P^H$.

Input Parameters

compq	CHARACTER*1. Must be 'V' or 'N'. If *compq* = 'V', then the Schur vectors (*Q*) are updated. If *compq* = 'N', then no Schur vectors are updated.
n	INTEGER. The order of the matrix *T* ($n \geq 0$).
t, *q*	REAL for strexc DOUBLE PRECISION for dtrexc COMPLEX for ctrexc DOUBLE COMPLEX for ztrexc. Arrays: *t*(*ldt*,*) contains the *n*-by-*n* matrix *T*. The second dimension of *t* must be at least max(1, *n*). *q*(*ldq*,*) If *compq* = 'V', then *q* must contain *Q* (Schur vectors). If *compq* = 'N', then *q* is not referenced. The second dimension of *q* must be at least max(1, *n*) if *compq* = 'V' and at least 1 if *compq* = 'N'.
ldt	INTEGER. The leading dimension of *t*; at least max(1, *n*).
ldq	INTEGER. The leading dimension of *q*; If *compq* = 'N', then $ldq \geq 1$. If *compq* = 'V', then $ldq \geq \max(1, n)$.
ifst, *ilst*	INTEGER. $1 \leq ifst \leq n$; $1 \leq ilst \leq n$. Must specify the reordering of the diagonal elements (or blocks, which is possible for real flavors) of the matrix *T*. The element (or block) with row index *ifst* is moved to row *ilst* by a sequence of exchanges between adjacent elements (or blocks).
work	REAL for strexc DOUBLE PRECISION for dtrexc. Array, size at least max (1, *n*).

Output Parameters

t	Overwritten by the updated matrix *S*.
q	If *compq* = 'V', *q* contains the updated matrix of Schur vectors.
ifst, *ilst*	Overwritten for real flavors only. If *ifst* pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; *ilst* always points to the first row of the block in its final position (which may differ from its input value by ±1).

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine trexc interface are the following:

t	Holds the matrix *T* of size (*n*,*n*).
q	Holds the matrix *Q* of size (*n*,*n*).
compq	Restored based on the presence of the argument *q* as follows: *compq* = 'V', if *q* is present, *compq* = 'N', if *q* is omitted.

Application Notes

The computed matrix *S* is exactly similar to a matrix $T+E$, where $||E||_2 = O(\varepsilon)*||T||_2$, and ε is the machine precision.

Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real.

The approximate number of floating-point operations is

for real flavors:	$6n(ifst-ilst)$ if *compq* = 'N'; $12n(ifst-ilst)$ if *compq* = 'V';
for complex flavors:	$20n(ifst-ilst)$ if *compq* = 'N'; $40n(ifst-ilst)$ if *compq* = 'V'.

?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers for the selected cluster of eigenvalues and respective invariant subspace.

Syntax

call strsen(*job*, *compq*, *select*, *n*, *t*, *ldt*, *q*, *ldq*, *wr*, *wi*, *m*, *s*, *sep*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call dtrsen(*job*, *compq*, *select*, *n*, *t*, *ldt*, *q*, *ldq*, *wr*, *wi*, *m*, *s*, *sep*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call ctrsen(*job*, *compq*, *select*, *n*, *t*, *ldt*, *q*, *ldq*, *w*, *m*, *s*, *sep*, *work*, *lwork*, *info*)

call ztrsen(*job*, *compq*, *select*, *n*, *t*, *ldt*, *q*, *ldq*, *w*, *m*, *s*, *sep*, *work*, *lwork*, *info*)

call trsen(*t*, *select* [,*wr*] [,*wi*] [,*m*] [,*s*] [,*sep*] [,*q*] [,*info*])

```
call trsen(t, select [,w] [,m] [,s] [,sep] [,q] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reorders the Schur factorization of a general matrix $A = Q*T*Q^T$ (for real flavors) or $A = Q*T*Q^H$ (for complex flavors) so that a selected cluster of eigenvalues appears in the leading diagonal elements (or, for real flavors, diagonal blocks) of the Schur form. The reordered Schur form R is computed by a unitary (orthogonal) similarity transformation: $R = Z^H*T*Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q*Z$, giving $A = P*R*P^H$.

Let

$$R = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{13} \end{bmatrix}$$

where the selected eigenvalues are precisely the eigenvalues of the leading m-by-m submatrix T_{11}. Let P be correspondingly partitioned as (Q_1Q_2) where Q_1 consists of the first m columns of Q. Then $A*Q_1 = Q_1*T_{11}$, and so the m columns of Q_1 form an orthonormal basis for the invariant subspace corresponding to the selected cluster of eigenvalues.

Optionally the routine also computes estimates of the reciprocal condition numbers of the average of the cluster of eigenvalues and of the invariant subspace.

Input Parameters

job

CHARACTER*1. Must be 'N' or 'E' or 'V' or 'B'.

If *job* = 'N', then no condition numbers are required.

If *job* = 'E', then only the condition number for the cluster of eigenvalues is computed.

If *job* = 'V', then only the condition number for the invariant subspace is computed.

If *job* = 'B', then condition numbers for both the cluster and the invariant subspace are computed.

compq

CHARACTER*1. Must be 'V' or 'N'.

If *compq* = 'V', then Q of the Schur vectors is updated.

If *compq* = 'N', then no Schur vectors are updated.

select — LOGICAL.

Array, size at least max (1, *n*).

Specifies the eigenvalues in the selected cluster. To select an eigenvalue λ_j, *select*(*j*) must be .TRUE.

For real flavors: to select a complex conjugate pair of eigenvalues λ_j and λ_{j+1} (corresponding 2 by 2 diagonal block), *select*(*j*) and/or *select*(*j* + 1) must be .TRUE.; the complex conjugate $\lambda_j$and λ_{j+1} must be either both included in the cluster or both excluded.

n — INTEGER. The order of the matrix *T* (*n*≥ 0).

t, *q*, *work* — REAL for strsen

DOUBLE PRECISION for dtrsen

COMPLEX for ctrsen

DOUBLE COMPLEX for ztrsen.

Arrays:

t(*ldt*,*) Theupper quasi-triangular *n*-by-*n* matrix *T*, in Schur canonical form.

The second dimension of *t* must be at least max(1, *n*).

q(*ldq*,*)

If *compq* = 'V', then *q* must contain the matrix *Q* of Schur vectors.

If *compq* = 'N', then *q* is not referenced.

The second dimension of *q* must be at least max(1, *n*) if *compq* = 'V' and at least 1 if *compq* = 'N'.

work is a workspace array, its dimension max(1, *lwork*).

ldt — INTEGER. The leading dimension of *t*; at least max(1, *n*).

ldq — INTEGER. The leading dimension of *q*;

If *compq* = 'N', then *ldq*≥ 1.

If *compq* = 'V', then *ldq*≥ max(1, *n*).

lwork — INTEGER. The dimension of the array *work*.

If *job* = 'V' or 'B', *lwork*≥ max(1,2*m*(*n*−*m*)).

If *job* = 'E', then *lwork*≥ max(1, *m*(*n*−*m*))

If *job* = 'N', then *lwork*≥ 1 for complex flavors and *lwork*≥ max(1,*n*) for real flavors.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for details.

iwork — INTEGER.*iwork*(*liwork*) is a workspace array. The array *iwork* is not referenced if *job* = 'N' or 'E'.

The actual amount of workspace required cannot exceed $n^2/2$ if *job* = 'V' or 'B'.

liwork

INTEGER.

The dimension of the array *iwork*.

If *job* = 'V' or 'B', *liwork* ≥ max(1,2*m*(*n*-*m*)).

If *job* = 'E' or 'E', *liwork* ≥ 1.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

t

Overwritten by the reordered matrix *R* in Schur canonical form with the selected eigenvalues in the leading diagonal blocks.

q

If *compq* = 'V', *q* contains the updated matrix of Schur vectors; the first m columns of the *Q* form an orthogonal basis for the specified invariant subspace.

w

COMPLEX for ctrsen

DOUBLE COMPLEX for ztrsen.

Array, size at least max(1, *n*). The recorded eigenvalues of *R*. The eigenvalues are stored in the same order as on the diagonal of *R*.

wr, *wi*

REAL for strsen

DOUBLE PRECISION for dtrsen

Arrays, size at least max(1, *n*). Contain the real and imaginary parts, respectively, of the reordered eigenvalues of *R*. The eigenvalues are stored in the same order as on the diagonal of *R*. Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

m

INTEGER.

For complex flavors: the dimension of the specified invariant subspaces, which is the same as the number of selected eigenvalues (see *select*).

For real flavors: the dimension of the specified invariant subspace. The value of *m* is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues (see *select*).

Constraint: 0 ≤*m*≤*n*.

s

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

If *job* = 'E' or 'B', *s* is a lower bound on the reciprocal condition number of the average of the selected cluster of eigenvalues.

If *m* = 0 or *n*, then *s* = 1.

For real flavors: if *info* = 1, then *s* is set to zero.*s* is not referenced if *job* = 'N' or 'V'.

sep — REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

If *job* = 'V' or 'B', *sep* is the estimated reciprocal condition number of the specified invariant subspace.

If *m* = 0 or *n*, then *sep* = |*T*|.

For real flavors: if *info* = 1, then *sep* is set to zero.

sep is not referenced if *job* = 'N' or 'E'.

work(1) — On exit, if *info* = 0, then *work*(1) returns the optimal size of *lwork*.

iwork(1) — On exit, if *info* = 0, then *iwork*(1) returns the optimal size of *liwork*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, the reordering of *T* failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); *T* may have been partially reordered, and *wr* and *wi* contain the eigenvalues in the same order as in *T*; *s* and *sep* (if requested) are set to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine trsen interface are the following:

t — Holds the matrix *T* of size (*n*,*n*).

select — Holds the vector of length *n*.

wr — Holds the vector of length *n*. Used in real flavors only.

wi — Holds the vector of length *n*. Used in real flavors only.

w — Holds the vector of length *n*. Used in complex flavors only.

q — Holds the matrix *Q* of size (*n*,*n*).

compq — Restored based on the presence of the argument *q* as follows: *compq* = 'V', if *q* is present, *compq* = 'N', if *q* is omitted.

job — Restored based on the presence of arguments *s* and sep as follows:

job = 'B', if both *s* and *sep* are present,

job = 'E', if *s* is present and *sep* omitted,

job = 'V', if *s* is omitted and *sep* present,

job = 'N', if both *s* and *sep* are omitted.

Application Notes

The computed matrix *R* is exactly similar to a matrix $T+E$, where $||E||_2 = O(\varepsilon)*||T||_2$, and ε is the machine precision. The computed *s* cannot underestimate the true reciprocal condition number by more than a factor of $(\min(m, n-m))_{1/2}$; *sep* may differ from the true value by $(m*n-m^2)_{1/2}$. The angle between the computed invariant subspace and the true subspace is $O(\varepsilon)*||A||_2/sep$. Note that if a 2-by-2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2-by-2 block to break into two 1-by-1 blocks, that is, for a pair of complex eigenvalues to become purely real.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set $lwork = -1$ ($liwork = -1$).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value ($work(1)$, $iwork(1)$) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?trsyl

Solves Sylvester equation for real quasi-triangular or complex triangular matrices.

Syntax

```
call strsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call dtrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ctrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ztrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call trsyl(a, b, c, scale [, trana] [,tranb] [,isgn] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves the Sylvester matrix equation $op(A)*X \pm X*op(B) = \alpha*C$, where $op(A) = A$ or A^H, and the matrices *A* and *B* are upper triangular (or, for real flavors, upper quasi-triangular in canonical Schur form); $\alpha \leq 1$ is a scale factor determined by the routine to avoid overflow in *X*; *A* is *m*-by-*m*, *B* is *n*-by-*n*, and *C* and *X* are both *m*-by-*n*. The matrix *X* is obtained by a straightforward process of back substitution.

The equation has a unique solution if and only if $\alpha_i \pm \beta_i \neq 0$, where $\{\alpha_i\}$ and $\{\beta_i\}$ are the eigenvalues of *A* and *B*, respectively, and the sign (+ or -) is the same as that used in the equation to be solved.

Input Parameters

trana

CHARACTER*1. Must be 'N' or 'T' or 'C'.

If $trana = 'N'$, then $op(A) = A$.

If *trana* = 'T', then op(*A*) = A^T (real flavors only).

If *trana* = 'C' then op(*A*) = A^H.

tranb CHARACTER*1. Must be 'N' or 'T' or 'C'.

If *tranb* = 'N', then op(*B*) = *B*.

If *tranb* = 'T', then op(*B*) = B^T (real flavors only).

If *tranb* = 'C', then op(*B*) = B^H.

isgn INTEGER. Indicates the form of the Sylvester equation.

If *isgn* = +1, op(*A*)***X* + *X**op(*B*) = alpha**C*.

If *isgn* = -1, op(*A*)***X* - *X**op(*B*) = alpha**C*.

m INTEGER. The order of *A*, and the number of rows in *X* and *C* (*m*≥ 0).

n INTEGER. The order of *B*, and the number of columns in *X* and *C* (*n*≥ 0).

a, *b*, *c* REAL for strsyl

DOUBLE PRECISION for dtrsyl

COMPLEX for ctrsyl

DOUBLE COMPLEX for ztrsyl.

Arrays:

a(*lda*,*) contains the matrix *A*.

The second dimension of *a* must be at least max(1, *m*).

b(*ldb*,*) contains the matrix *B*.

The second dimension of *b* must be at least max(1, *n*).

c(*ldc*,*) contains the matrix *C*.

The second dimension of *c* must be at least max(1, *n*).

lda INTEGER. The leading dimension of *a*; at least max(1, *m*).

ldb INTEGER. The leading dimension of *b*; at least max(1, *n*).

ldc INTEGER. The leading dimension of *c*; at least max(1, *m*) .

Output Parameters

c Overwritten by the solution matrix *X*.

scale REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

The value of the scale factor α.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, *A* and *B* have common or close eigenvalues; perturbed values were used to solve the equation.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `trsyl` interface are the following:

a	Holds the matrix *A* of size (*m*,*m*).
b	Holds the matrix *B* of size (*n*,*n*).
c	Holds the matrix *C* of size (*m*,*n*).
trana	Must be 'N', 'C', or 'T'. The default value is 'N'.
tranb	Must be 'N', 'C', or 'T'. The default value is 'N'.
isgn	Must be +1 or -1. The default value is +1.

Application Notes

Let *X* be the exact, *Y* the corresponding computed solution, and *R* the residual matrix: $R = C - (AY \pm YB)$. Then the residual is always small:

$||R||_F = O(\varepsilon) * (||A||_F + ||B||_F) * ||Y||_F$.

However, *Y* is not necessarily the exact solution of a slightly perturbed equation; in other words, the solution is not backwards stable.

For the forward error, the following bound holds:

$||Y - X||_F \leq ||R||_F / \mathrm{sep}(A, B)$

but this may be a considerable overestimate. See [Golub96] for a definition of sep(*A*, *B*).

The approximate number of floating-point operations for real flavors is $m*n*(m + n)$. For complex flavors it is 4 times greater.

Generalized Nonsymmetric Eigenvalue Problems: LAPACK Computational Routines

This topic describes LAPACK routines for solving generalized nonsymmetric eigenvalue problems, reordering the generalized Schur factorization of a pair of matrices, as well as performing a number of related computational tasks.

A *generalized nonsymmetric eigenvalue problem* is as follows: given a pair of nonsymmetric (or non-Hermitian) *n*-by-*n* matrices *A* and *B*, find the *generalized eigenvalues*λ and the corresponding *generalized eigenvectors* *x* and *y* that satisfy the equations

$Ax = \lambda Bx$ (right generalized eigenvectors *x*)

and

$y^H A = \lambda y^H B$ (left generalized eigenvectors *y*).

Table "Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems" lists LAPACK routines (FORTRAN 77 interface) used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester equation. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems

Routine name	Operation performed
gghrd	Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.
ggbal	Balances a pair of general real or complex matrices.
ggbak	Forms the right or left eigenvectors of a generalized eigenvalue problem.
gghd3	Reduces a pair of matrices to generalized upper Hessenberg form.
hgeqz	Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).
tgevc	Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices
tgexc	Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.
tgsen	Reorders the *generalized* Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).
tgsyl	Solves the generalized Sylvester equation.
tgsyl	Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

?gghrd
Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.

Syntax

```
call sgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call dgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call cgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call zgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call gghrd(a, b [,ilo] [,ihi] [,q] [,z] [,compq] [,compz] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a pair of real/complex matrices (*A*,*B*) to generalized upper Hessenberg form using orthogonal/unitary transformations, where *A* is a general matrix and *B* is upper triangular. The form of the generalized eigenvalue problem is $A*x = \lambda*B*x$, and *B* is typically made upper triangular by computing its *QR* factorization and moving the orthogonal matrix *Q* to the left side of the equation.

This routine simultaneously reduces *A* to a Hessenberg matrix *H*:

$Q^H*A*Z = H$

and transforms *B* to another upper triangular matrix *T*:

$Q^H*B*Z = T$

in order to reduce the problem to its standard form $H*y = \lambda*T*y$, where $y = Z^H*x$.

The orthogonal/unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q_1 and Z_1, so that

$Q_1*A*Z_1^H = (Q_1*Q)*H*(Z_1*Z)^H$

$Q_1*B*Z_1^H = (Q_1*Q)*T*(Z_1*Z)^H$

If Q_1 is the orthogonal/unitary matrix from the QR factorization of B in the original equation $A*x = \lambda*B*x$, then the routine ?gghrd reduces the original problem to generalized Hessenberg form.

Input Parameters

compq

CHARACTER*1. Must be 'N', 'I', or 'V'.

If *compq* = 'N', matrix Q is not computed.

If *compq* = 'I', Q is initialized to the unit matrix, and the orthogonal/unitary matrix Q is returned;

If *compq* = 'V', Q must contain an orthogonal/unitary matrix Q_1 on entry, and the product Q_1*Q is returned.

compz

CHARACTER*1. Must be 'N', 'I', or 'V'.

If *compz* = 'N', matrix Z is not computed.

If *compz* = 'I', Z is initialized to the unit matrix, and the orthogonal/unitary matrix Z is returned;

If *compz* = 'V', Z must contain an orthogonal/unitary matrix Z_1 on entry, and the product Z_1*Z is returned.

n

INTEGER. The order of the matrices A and B ($n \geq 0$).

ilo, *ihi*

INTEGER. *ilo* and *ihi* mark the rows and columns of A which are to be reduced. It is assumed that A is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*. Values of *ilo* and *ihi* are normally set by a previous call to ggbal; otherwise they should be set to 1 and *n* respectively.

Constraint:

If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;

if $n = 0$, then $ilo = 1$ and $ihi = 0$.

a, *b*, *q*, *z*

REAL for sgghrd

DOUBLE PRECISION for dgghrd

COMPLEX for cgghrd

DOUBLE COMPLEX for zgghrd.

Arrays:

a(*lda*,*) contains the *n*-by-*n* general matrix A.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the *n*-by-*n* upper triangular matrix B.

The second dimension of *b* must be at least max(1, *n*).

q(*ldq*,*)

If *compq* = 'N', then *q* is not referenced.

If *compq* = 'V', then *q* must contain the orthogonal/unitary matrix Q_1, typically from the *QR* factorization of *B*.

The second dimension of *q* must be at least max(1, *n*).

z(*ldz*,*)

If *compz* = 'N', then *z* is not referenced.

If *compz* = 'V', then *z* must contain the orthogonal/unitary matrix Z_1.

The second dimension of *z* must be at least max(1, *n*).

lda INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldb INTEGER. The leading dimension of *b*; at least max(1, *n*).

ldq INTEGER. The leading dimension of *q*;

If *compq* = 'N', then *ldq*≥ 1.

If *compq* = 'I'or 'V', then *ldq*≥ max(1, *n*).

ldz INTEGER. The leading dimension of *z*;

If *compz* = 'N', then *ldz*≥ 1.

If *compz* = 'I'or 'V', then *ldz*≥ max(1, *n*).

Output Parameters

a On exit, the upper triangle and the first subdiagonal of *A* are overwritten with the upper Hessenberg matrix *H*, and the rest is set to zero.

b On exit, overwritten by the upper triangular matrix $T = Q^H*B*Z$. The elements below the diagonal are set to zero.

q If *compq* = 'I', then *q* contains the orthogonal/unitary matrix *Q*, ;

If *compq* = 'V', then *q* is overwritten by the product Q_1*Q.

z If *compz* = 'I', then *z* contains the orthogonal/unitary matrix *Z*;

If *compz* = 'V', then *z* is overwritten by the product Z_1*Z.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gghrd interface are the following:

a Holds the matrix *A* of size (*n*,*n*).

b	Holds the matrix *B* of size (*n*,*n*).
q	Holds the matrix *Q* of size (*n*,*n*).
z	Holds the matrix *Z* of size (*n*,*n*).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
compq	If omitted, this argument is restored based on the presence of argument *q* as follows: *compq* = 'I', if *q* is present, *compq* = 'N', if *q* is omitted. If present, *compq* must be equal to 'I' or 'V' and the argument *q* must also be present. Note that there will be an error condition if *compq* is present and *q* omitted.
compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = 'I', if *z* is present, *compz* = 'N', if *z* is omitted. If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted.

?ggbal

Balances a pair of general real or complex matrices.

Syntax

```
call sggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call dggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call cggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call zggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call ggbal(a, b [,ilo] [,ihi] [,lscale] [,rscale] [,job] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine balances a pair of general real/complex matrices (*A*,*B*). This involves, first, permuting *A* and *B* by similarity transformations to isolate eigenvalues in the first 1 to *ilo*-1 and last *ihi*+1 to *n* elements on the diagonal;and second, applying a diagonal similarity transformation to rows and columns *ilo* to *ihi* to make the rows and columns as close in norm as possible. Both steps are optional. Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem *A*x* = λ*B*x*.

Input Parameters

job	CHARACTER*1. Specifies the operations to be performed on *A* and *B*. Must be 'N' or 'P' or 'S' or 'B'. If *job* = 'N ', then no operations are done; simply set *ilo* =1, *ihi*=*n*, *lscale*(*i*) =1.0 and *rscale*(*i*)=1.0 for

	i = 1,..., *n*.
	If *job* = 'P', then permute only.
	If *job* = 'S', then scale only.
	If *job* = 'B', then both permute and scale.
n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
a, *b*	REAL for sggbal
	DOUBLE PRECISION for dggbal
	COMPLEX for cggbal
	DOUBLE COMPLEX for zggbal.
	Arrays:
	a(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).
	b(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *n*).
	If *job* = 'N', *a* and *b* are not referenced.
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; at least max(1, *n*).
work	REAL for single precision flavors
	DOUBLE PRECISION for double precision flavors.
	Workspace array, size at least max(1, 6*n*) when *job* = 'S'or 'B', or at least 1 when *job* = 'N'or 'P'.

Output Parameters

a, *b*	Overwritten by the balanced matrices *A* and *B*, respectively.
ilo, *ihi*	INTEGER. *ilo* and *ihi* are set to integers such that on exit $A_{i,j} = 0$ and $B_{i,j} = 0$ if *i*>*j* and *j*=1,...,*ilo*-1 or *i*=*ihi*+1,..., *n*.
	If *job* = 'N'or 'S', then *ilo* = 1 and *ihi* = *n*.
lscale, *rscale*	REAL for single precision flavors
	DOUBLE PRECISION for double precision flavors.
	Arrays, size at least max(1, *n*).
	lscale contains details of the permutations and scaling factors applied to the left side of *A* and *B*.
	If P_j is the index of the row interchanged with row *j*, and D_j is the scaling factor applied to row *j*, then
	lscale(*j*) = P_j, for *j* = 1,..., *ilo*-1
	= D_j, for *j* = *ilo*,...,*ihi*
	= P_j, for *j* = *ihi*+1,..., *n*.

rscale contains details of the permutations and scaling factors applied to the right side of *A* and *B*.

If P_j is the index of the column interchanged with column *j*, and D_j is the scaling factor applied to column *j*, then

`rscale(j) =` P_j, for `j = 1,..., ilo-1`

`=` D_j, for `j = ilo,...,ihi`

`=` P_j, for `j = ihi+1,..., n`

The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

info — INTEGER.

If `info = 0`, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `ggbal` interface are the following:

a — Holds the matrix *A* of size (*n*,*n*).

b — Holds the matrix *B* of size (*n*,*n*).

lscale — Holds the vector of length (*n*).

rscale — Holds the vector of length (*n*).

ilo — Default value for this argument is `ilo = 1`.

ihi — Default value for this argument is `ihi = n`.

job — Must be `'B'`, `'S'`, `'P'`, or `'N'`. The default value is `'B'`.

?ggbak

Forms the right or left eigenvectors of a generalized eigenvalue problem.

Syntax

```
call sggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call dggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call cggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call zggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call ggbak(v [, ilo] [,ihi] [,lscale] [,rscale] [,job] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine forms the right or left eigenvectors of a real/complex generalized eigenvalue problem

$A*x = \lambda*B*x$

by backward transformation on the computed eigenvectors of the balanced pair of matrices output by ggbal.

Input Parameters

job

CHARACTER*1. Specifies the type of backward transformation required. Must be 'N', 'P', 'S', or 'B'.

If *job* = 'N', then no operations are done; return.

If *job* = 'P', then do backward transformation for permutation only.

If *job* = 'S', then do backward transformation for scaling only.

If *job* = 'B', then do backward transformation for both permutation and scaling. This argument must be the same as the argument *job* supplied to ?ggbal.

side

CHARACTER*1. Must be 'L' or 'R'.

If *side* = 'L', then *v* contains left eigenvectors.

If *side* = 'R', then *v* contains right eigenvectors.

n

INTEGER. The number of rows of the matrix *V* ($n \geq 0$).

ilo, *ihi*

INTEGER. The integers *ilo* and *ihi* determined by ?gebal. Constraint:

If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;

if $n = 0$, then $ilo = 1$ and $ihi = 0$.

lscale, *rscale*

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays, size at least max(1, *n*).

The array *lscale* contains details of the permutations and/or scaling factors applied to the left side of *A* and *B*, as returned by ?ggbal.

The array *rscale* contains details of the permutations and/or scaling factors applied to the right side of *A* and *B*, as returned by ?ggbal.

m

INTEGER. The number of columns of the matrix *V*

($m \geq 0$).

v

REAL for sggbak

DOUBLE PRECISION for dggbak

COMPLEX for cggbak

DOUBLE COMPLEX for zggbak.

Array *v*(*ldv*,*) . Contains the matrix of right or left eigenvectors to be transformed, as returned by tgevc.

The second dimension of *v* must be at least max(1, *m*).

ldv

INTEGER. The leading dimension of *v*; at least max(1, *n*) .

Output Parameters

v	Overwritten by the transformed eigenvectors
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ggbak interface are the following:

v	Holds the matrix *V* of size (*n*,*m*).
lscale	Holds the vector of length *n*.
rscale	Holds the vector of length *n*.
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
job	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.
side	If omitted, this argument is restored based on the presence of arguments *lscale* and *rscale* as follows: *side* = 'L', if *lscale* is present and *rscale* omitted, *side* = 'R', if *lscale* is omitted and *rscale* present. Note that there will be an error condition if both *lscale* and *rscale* are present or if they both are omitted.

?gghd3

Reduces a pair of matrices to generalized upper Hessenberg form.

Syntax

```
call sgghd3 (compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, work, lwork, info )
call dgghd3 (compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, work, lwork, info )
call cgghd3 (compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, work, lwork, info )
call zgghd3 (compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, work, lwork, info )
```

Include Files

- mkl.fi

Description

?gghd3 reduces a pair of real or complex matrices (*A*, *B*) to generalized upper Hessenberg form using orthogonal/unitary transformations, where *A* is a general matrix and *B* is upper triangular. The form of the generalized eigenvalue problem is

$A*x = \lambda*B*x$,

and *B* is typically made upper triangular by computing its QR factorization and moving the orthogonal/unitary matrix *Q* to the left side of the equation.

This subroutine simultaneously reduces *A* to a Hessenberg matrix *H*:

$Q^T*A*Z = H$ for real flavors

or

$Q^T*A*Z = H$ for complex flavors

and transforms *B* to another upper triangular matrix *T*:

$Q^T*B*Z = T$ for real flavors

or

$Q^T*B*Z = T$ for complex flavors

in order to reduce the problem to its standard form

$H*y = \lambda*T*y$

where $y = Z^T*x$ for real flavors

or

$y = Z^T*x$ for complex flavors.

The orthogonal/unitary matrices *Q* and *Z* are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q_1 and Z_1, so that

for real flavors:

$Q_1 * A * Z_1^T = (Q_1*Q) * H * (Z_1*Z)^T$

$Q_1 * B * Z_1^T = (Q_1*Q) * T * (Z_1*Z)^T$

for complex flavors:

$Q_1 * A * Z_1^H = (Q_1*Q) * H * (Z_1*Z)^T$

$Q_1 * B * Z_1^T = (Q_1*Q) * T * (Z_1*Z)^T$

If Q_1 is the orthogonal/unitary matrix from the QR factorization of *B* in the original equation $A*x = \lambda*B*x$, then ?gghd3 reduces the original problem to generalized Hessenberg form.

This is a blocked variant of ?gghrd, using matrix-matrix multiplications for parts of the computation to enhance performance.

Input Parameters

compq

CHARACTER*1. = 'N': do not compute *q*;

= 'I': *q* is initialized to the unit matrix, and the orthogonal/unitary matrix *Q* is returned;

= 'V': *q* must contain an orthogonal/unitary matrix Q_1 on entry, and the product Q_1*q is returned.

compz

CHARACTER*1. = 'N': do not compute *z*;

= 'I': *z* is initialized to the unit matrix, and the orthogonal/unitary matrix *Z* is returned;

= 'V': *z* must contain an orthogonal/unitary matrix Z_1 on entry, and the product Z_1*z is returned.

n

INTEGER. The order of the matrices *A* and *B*.

n≥ 0.

ilo, *ihi*

INTEGER. *ilo* and *ihi* mark the rows and columns of *a* which are to be reduced. It is assumed that *a* is already upper triangular in rows and columns 1:*ilo* - 1 and *ihi* + 1:*n*. *ilo* and *ihi* are normally set by a previous call to ?ggbal; otherwise they should be set to 1 and *n*, respectively.

1 ≤*ilo*≤*ihi*≤*n*, if *n* > 0; *ilo*=1 and *ihi*=0, if *n*=0.

a

REAL for sgghd3

DOUBLE PRECISION for dgghd3

COMPLEX for cgghd3

DOUBLE COMPLEX for zgghd3

Array, size (*lda*, *n*).

On entry, the *n*-by-*n* general matrix to be reduced.

lda

INTEGER. The leading dimension of the array *a*.

lda≥ max(1,*n*).

b

REAL for sgghd3

DOUBLE PRECISION for dgghd3

COMPLEX for cgghd3

DOUBLE COMPLEX for zgghd3

Array, (*ldb*, *n*).

On entry, the*n*-by-*n* upper triangular matrix *B*.

ldb

INTEGER. The leading dimension of the array *b*.

ldb≥ max(1,*n*).

q

REAL for sgghd3

DOUBLE PRECISION for dgghd3

COMPLEX for cgghd3

DOUBLE COMPLEX for zgghd3

Array, size (*ldq*, *n*).

On entry, if *compq* = 'V', the orthogonal/unitary matrix Q_1, typically from the QR factorization of *b*.

ldq

INTEGER. The leading dimension of the array *q*.

ldq≥*n* if *compq*='V' or 'I'; *ldq*≥ 1 otherwise.

z

REAL for sgghd3

DOUBLE PRECISION for dgghd3

COMPLEX for cgghd3

DOUBLE COMPLEX for zgghd3

Array, size (*ldz*, *n*).

On entry, if *compz* = 'V', the orthogonal/unitary matrix Z_1.

Not referenced if *compz*='N'.

ldz

INTEGER. The leading dimension of the array *z*. *ldz*≥*n* if *compz*='V' or 'I'; *ldz*≥ 1 otherwise.

work

REAL for sgghd3

DOUBLE PRECISION for dgghd3

COMPLEX for cgghd3

DOUBLE COMPLEX for zgghd3

Array, size (*lwork*)

lwork

INTEGER. The length of the array work.

lwork≥ 1.

For optimum performance *lwork*≥ 6**n***NB*, where *NB* is the optimal blocksize.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

a

On exit, the upper triangle and the first subdiagonal of *a* are overwritten with the upper Hessenberg matrix *H*, and the rest is set to zero.

b

On exit, the upper triangular matrix $T = Q^T BZ$ for real flavors or $T = Q^H BZ$ for complex flavors. The elements below the diagonal are set to zero.

q

On exit, if *compq*='I', the orthogonal/unitary matrix *Q*, and if *compq* = 'V', the product Q_1**Q*.

Not referenced if *compq*='N'.

z

On exit, if *compz*='I', the orthogonal/unitary matrix *Z*, and if *compz* = 'V', the product Z_1**Z*.

Not referenced if *compz*='N'.

work

On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

info

INTEGER. = 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

Application Notes

This routine reduces *A* to Hessenberg form and maintains *B* in using a blocked variant of Moler and Stewart's original algorithm, as described by Kagstrom, Kressner, Quintana-Orti, and Quintana-Orti (BIT 2008).

?hgeqz
Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).

Syntax

`call shgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphar, alphai, beta, q, ldq, z, ldz, work, lwork, info)`

`call dhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphar, alphai, beta, q, ldq, z, ldz, work, lwork, info)`

`call chgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q, ldq, z, ldz, work, lwork, rwork, info)`

`call zhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q, ldq, z, ldz, work, lwork, rwork, info)`

`call hgeqz(h, t [,ilo] [,ihi] [,alphar] [,alphai] [,beta] [,q] [,z] [,job] [,compq] [,compz] [,info])`

`call hgeqz(h, t [,ilo] [,ihi] [,alpha] [,beta] [,q] [,z] [,job] [,compq] [, compz] [,info])`

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes the eigenvalues of a real/complex matrix pair (*H*,*T*), where *H* is an upper Hessenberg matrix and *T* is upper triangular, using the double-shift version (for real flavors) or single-shift version (for complex flavors) of the *QZ* method. Matrix pairs of this type are produced by the reduction to generalized upper Hessenberg form of a real/complex matrix pair (*A*,*B*):

$A = Q_1 * H * Z_1^H$, $B = Q_1 * T * Z_1^H$,

as computed by `?gghrd`.

For real flavors:

If *job* = `'S'`, then the Hessenberg-triangular pair (*H*,*T*) is reduced to generalized Schur form,

$H = Q * S * Z^T$, $T = Q * P * Z^T$,

where *Q* and *Z* are orthogonal matrices, *P* is an upper triangular matrix, and *S* is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks. The 1-by-1 blocks correspond to real eigenvalues of the matrix pair (*H*,*T*) and the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues.

Additionally, the 2-by-2 upper triangular diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* are reduced to positive diagonal form, that is, if $S_{j+1,j}$ is non-zero, then $P_{j+1,\,j} = P_{j,\,j+1} = 0$, $P_{j,\,j} > 0$, and $P_{j+1,\,j+1} > 0$.

For complex flavors:

If *job* = `'S'`, then the Hessenberg-triangular pair (*H*,*T*) is reduced to generalized Schur form,

$H = Q^* S^* Z^H$, $T = Q^* P^* Z^H$,

where Q and Z are unitary matrices, and S and P are upper triangular.

For all function flavors:

Optionally, the orthogonal/unitary matrix Q from the generalized Schur factorization may be post-multiplied by an input matrix Q_1, and the orthogonal/unitary matrix Z may be post-multiplied by an input matrix Z_1.

If Q_1 and Z_1 are the orthogonal/unitary matrices from `?gghrd` that reduced the matrix pair (A,B) to generalized upper Hessenberg form, then the output matrices Q_1Q and Z_1Z are the orthogonal/unitary factors from the generalized Schur factorization of (A,B):

$A = (Q_1Q)^* S^* (Z_1Z)^H$, $B = (Q_1Q)^* P^* (Z_1Z)^H$.

To avoid overflow, eigenvalues of the matrix pair (H,T) (equivalently, of (A,B)) are computed as a pair of values (*alpha*,*beta*). For `chgeqz`/`zhgeqz`, *alpha* and *beta* are complex, and for `shgeqz`/`dhgeqz`, *alpha* is complex and *beta* real. If *beta* is nonzero, $\lambda = alpha/beta$ is an eigenvalue of the generalized nonsymmetric eigenvalue problem (GNEP)

$A^*x = \lambda^*B^*x$

and if *alpha* is nonzero, $\mu = beta/alpha$ is an eigenvalue of the alternate form of the GNEP

$\mu^*A^*y = B^*y$.

Real eigenvalues (for real flavors) or the values of *alpha* and *beta* for the i-th eigenvalue (for complex flavors) can be read directly from the generalized Schur form:

$alpha = S_{i,\,i}$, $beta = P_{i,\,i}$.

Input Parameters

job

`CHARACTER*1`. Specifies the operations to be performed. Must be `'E'` or `'S'`.

If *job* = `'E'`, then compute eigenvalues only;

If *job* = `'S'`, then compute eigenvalues and the Schur form.

compq

`CHARACTER*1`. Must be `'N'`, `'I'`, or `'V'`.

If *compq* = `'N'`, left Schur vectors (q) are not computed;

If *compq* = `'I'`, q is initialized to the unit matrix and the matrix of left Schur vectors of (H,T) is returned;

If *compq* = `'V'`, q must contain an orthogonal/unitary matrix Q_1 on entry and the product Q_1*Q is returned.

compz

`CHARACTER*1`. Must be `'N'`, `'I'`, or `'V'`.

If *compz* = `'N'`, right Schur vectors (z) are not computed;

If *compz* = `'I'`, z is initialized to the unit matrix and the matrix of right Schur vectors of (H,T) is returned;

If *compz* = `'V'`, z must contain an orthogonal/unitary matrix Z_1 on entry and the product Z_1*Z is returned.

n

`INTEGER`. The order of the matrices H, T, Q, and Z

($n \geq 0$).

ilo, *ihi*

INTEGER. *ilo* and *ihi* mark the rows and columns of *H* which are in Hessenberg form. It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*.

Constraint:

If $n > 0$, then $1 \le ilo \le ihi \le n$;

if $n = 0$, then $ilo = 1$ and $ihi = 0$.

h, *t*, *q*, *z*, *work*

REAL for shgeqz

DOUBLE PRECISION for dhgeqz

COMPLEX for chgeqz

DOUBLE COMPLEX for zhgeqz.

Arrays:

On entry, *h*(*ldh*,*) contains the *n*-by-*n* upper Hessenberg matrix *H*.

The second dimension of *h* must be at least max(1, *n*).

On entry, *t*(*ldt*,*) contains the *n*-by-*n* upper triangular matrix *T*.

The second dimension of *t* must be at least max(1, *n*).

q(*ldq*,*) :

On entry, if *compq* = 'V', this array contains the orthogonal/unitary matrix Q_1 used in the reduction of (*A*,*B*) to generalized Hessenberg form.

If *compq* = 'N', then *q* is not referenced.

The second dimension of *q* must be at least max(1, *n*).

z(*ldz*,*) :

On entry, if *compz* = 'V', this array contains the orthogonal/unitary matrix Z_1 used in the reduction of (*A*,*B*) to generalized Hessenberg form.

If *compz* = 'N', then *z* is not referenced.

The second dimension of *z* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

ldh

INTEGER. The leading dimension of *h*; at least max(1, *n*).

ldt

INTEGER. The leading dimension of *t*; at least max(1, *n*).

ldq

INTEGER. The leading dimension of *q*;

If *compq* = 'N', then $ldq \ge 1$.

If *compq* = 'I' or 'V', then $ldq \ge \max(1, n)$.

ldz

INTEGER. The leading dimension of *z*;

If *compq* = 'N', then $ldz \ge 1$.

If *compq* = 'I' or 'V', then $ldz \ge \max(1, n)$.

lwork

INTEGER. The dimension of the array *work*.

$lwork \ge \max(1, n)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for details.

rwork

REAL for chgeqz

DOUBLE PRECISION for zhgeqz.

Workspace array, size at least max(1, *n*). Used in complex flavors only.

Output Parameters

h

For real flavors:

If *job* = 'S', then on exit *h* contains the upper quasi-triangular matrix *S* from the generalized Schur factorization.

If *job* = 'E', then on exit the diagonal blocks of *h* match those of *S*, but the rest of *h* is unspecified.

For complex flavors:

If *job* = 'S', then, on exit, *h* contains the upper triangular matrix *S* from the generalized Schur factorization.

If *job* = 'E', then on exit the diagonal of *h* matches that of *S*, but the rest of *h* is unspecified.

t

If *job* = 'S', then, on exit, *t* contains the upper triangular matrix *P* from the generalized Schur factorization.

For real flavors:

2-by-2 diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* are reduced to positive diagonal form, that is, if *h*(j+1,j) is non-zero, then *t*(j+1,j)=*t*(j,j+1)=0 and *t*(j,j) and *t*(j+1,j+1) will be positive.

If *job* = 'E', then on exit the diagonal blocks of *t* match those of *P*, but the rest of *t* is unspecified.

For complex flavors:

if *job* = 'E', then on exit the diagonal of *t* matches that of *P*, but the rest of *t* is unspecified.

alphar, *alphai*

REAL for shgeqz;

DOUBLE PRECISION for dhgeqz.

Arrays, size at least max(1, *n*). The real and imaginary parts, respectively, of each scalar *alpha* defining an eigenvalue of GNEP.

If *alphai*(*j*) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-th eigenvalues are a complex conjugate pair, with

alphai(*j*+1) = -*alphai*(*j*).

alpha

COMPLEX for chgeqz;

DOUBLE COMPLEX for zhgeqz.

Array, size at least max(1, *n*).

The complex scalars *alpha* that define the eigenvalues of GNEP. `alphai(i) = S_i, i` in the generalized Schur factorization.

beta

`REAL` for `shgeqz`

`DOUBLE PRECISION` for `dhgeqz`

`COMPLEX` for `chgeqz`

`DOUBLE COMPLEX` for `zhgeqz`.

Array, size at least max(1, *n*).

For real flavors:

The scalars *beta* that define the eigenvalues of GNEP.

Together, the quantities `alpha = (alphar(j), alphai(j))` and `beta = beta(j)` represent the *j*-th eigenvalue of the matrix pair (*A*,*B*), in one of the forms `lambda = alpha/beta` or `mu = beta/alpha`. Since either *lambda* or *mu* may overflow, they should not, in general, be computed.

For complex flavors:

The real non-negative scalars *beta* that define the eigenvalues of GNEP.

`beta(i) = P_i, i` in the generalized Schur factorization. Together, the quantities `alpha = alpha(j)` and `beta = beta(j)` represent the *j*-th eigenvalue of the matrix pair (*A*,*B*), in one of the forms `lambda = alpha/beta` or `mu = beta/alpha`. Since either *lambda* or *mu* may overflow, they should not, in general, be computed.

q

On exit, if `compq = 'I'`, *q* is overwritten by the orthogonal/unitary matrix of left Schur vectors of the pair (*H*,*T*), and if `compq = 'V'`, *q* is overwritten by the orthogonal/unitary matrix of left Schur vectors of (*A*,*B*).

z

On exit, if `compz = 'I'`, *z* is overwritten by the orthogonal/unitary matrix of right Schur vectors of the pair (*H*,*T*), and if `compz = 'V'`, *z* is overwritten by the orthogonal/unitary matrix of right Schur vectors of (*A*,*B*).

work(1)

If `info`≥ 0, on exit, `work(1)` contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

`INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = 1,..., n`, the *QZ* iteration did not converge.

(*H*,*T*) is not in Schur form, but *alphar*(*i*), *alphai*(*i*) (for real flavors), *alpha*(*i*) (for complex flavors), and *beta*(*i*), `i=info+1,..., n` should be correct.

If `info = n+1,...,2n`, the shift calculation failed.

(*H*,*T*) is not in Schur form, but *alphar*(*i*), *alphai*(*i*) (for real flavors), *alpha*(*i*) (for complex flavors), and *beta*(*i*), `i =info-n+1,..., n` should be correct.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hgeqz` interface are the following:

h	Holds the matrix *H* of size (*n*,*n*).
t	Holds the matrix *T* of size (*n*,*n*).
alphar	Holds the vector of length *n*. Used in real flavors only.
alphai	Holds the vector of length *n*. Used in real flavors only.
alpha	Holds the vector of length *n*. Used in complex flavors only.
beta	Holds the vector of length *n*.
q	Holds the matrix *Q* of size (*n*,*n*).
z	Holds the matrix *Z* of size (*n*,*n*).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
job	Must be 'E' or 'S'. The default value is 'E'.
compq	If omitted, this argument is restored based on the presence of argument *q* as follows: *compq* = 'I', if *q* is present, *compq* = 'N', if *q* is omitted. If present, *compq* must be equal to 'I' or 'V' and the argument *q* must also be present. Note that there will be an error condition if *compq* is present and *q* omitted.
compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = 'I', if *z* is present, *compz* = 'N', if *z* is omitted. If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note an error condition if *compz* is present and *z* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgevc

Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices.

Syntax

```
call stgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, info)
call dtgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, info)
call ctgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
call ztgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
call tgevc(s, p [,howmny] [,select] [,vl] [,vr] [,m] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes some or all of the right and/or left eigenvectors of a pair of real/complex matrices (*S*,*P*), where *S* is quasi-triangular (for real flavors) or upper triangular (for complex flavors) and *P* is upper triangular.

Matrix pairs of this type are produced by the generalized Schur factorization of a real/complex matrix pair (*A*,*B*):

$A = Q*S*Z^H$, $B = Q*P*Z^H$

as computed by ?gghrd plus ?hgeqz.

The right eigenvector *x* and the left eigenvector *y* of (*S*,*P*) corresponding to an eigenvalue *w* are defined by:

$S*x = w*P*x$, $y^H*S = w*y^H*P$

The eigenvalues are not input to this routine, but are computed directly from the diagonal blocks or diagonal elements of *S* and *P*.

This routine returns the matrices *X* and/or *Y* of right and left eigenvectors of (*S*,*P*), or the products *Z***X* and/or *Q***Y*, where *Z* and *Q* are input matrices.

If *Q* and *Z* are the orthogonal/unitary factors from the generalized Schur factorization of a matrix pair (*A*,*B*), then *Z***X* and *Q***Y* are the matrices of right and left eigenvectors of (*A*,*B*).

Input Parameters

side

CHARACTER*1. Must be 'R', 'L', or 'B'.

If *side* = 'R', compute right eigenvectors only.

If *side* = 'L', compute left eigenvectors only.

If *side* = 'B', compute both right and left eigenvectors.

howmny — CHARACTER*1. Must be 'A', 'B', or 'S'.

If *howmny* = 'A', compute all right and/or left eigenvectors.

If *howmny* = 'B', compute all right and/or left eigenvectors, backtransformed by the matrices in *vr* and/or *vl*.

If *howmny* = 'S', compute selected right and/or left eigenvectors, specified by the logical array *select*.

select — LOGICAL.

Array, size at least max (1, *n*).

If *howmny* = 'S', *select* specifies the eigenvectors to be computed.

If *howmny* = 'A'or 'B', *select* is not referenced.

For real flavors:

If *w*(*j*) is a real eigenvalue, the corresponding real eigenvector is computed if *select*(*j*) is .TRUE..

If *w*(*j*) and *omega*(*j* + 1) are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either *select*(*j*) or *select*(*j* + 1) is .TRUE., and on exit *select*(*j*) is set to .TRUE.and *select*(*j* + 1) is set to .FALSE..

For complex flavors:

The eigenvector corresponding to the *j*-th eigenvalue is computed if *select*(*j*) is .TRUE..

n — INTEGER. The order of the matrices *S* and *P* (*n*≥ 0).

s, *p*, *vl*, *vr*, *work* — REAL for stgevc

DOUBLE PRECISION for dtgevc

COMPLEX for ctgevc

DOUBLE COMPLEX for ztgevc.

Arrays:

s(*lds*,*) contains the matrix *S* from a generalized Schur factorization as computed by ?hgeqz. This matrix is upper quasi-triangular for real flavors, and upper triangular for complex flavors.

The second dimension of *s* must be at least max(1, *n*).

p(*ldp*,*) contains the upper triangular matrix *P* from a generalized Schur factorization as computed by ?hgeqz.

For real flavors, 2-by-2 diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* must be in positive diagonal form.

For complex flavors, *P* must have real diagonal elements. The second dimension of *p* must be at least max(1, *n*).

If *side* = 'L' or 'B' and *howmny* = 'B', *vl*(*ldvl*,*) must contain an *n*-by-*n* matrix *Q* (usually the orthogonal/unitary matrix *Q* of left Schur vectors returned by ?hgeqz). The second dimension of *vl* must be at least max(1, *mm*).

If *side* = 'R', *vl* is not referenced.

If *side* = 'R' or 'B' and *howmny* = 'B', *vr*(*ldvr*,*) must contain an *n*-by-*n* matrix *Z* (usually the orthogonal/unitary matrix *Z* of right Schur vectors returned by ?hgeqz). The second dimension of *vr* must be at least max(1, *mm*).

If *side* = 'L', *vr* is not referenced.

work(*) is a workspace array.

size at least max (1, 6**n*) for real flavors and at least max (1, 2**n*) for complex flavors.

lds INTEGER. The leading dimension of *s*; at least max(1, *n*).

ldp INTEGER. The leading dimension of *p*; at least max(1, *n*).

ldvl INTEGER. The leading dimension of *vl*;

If *side* = 'L' or 'B', then *ldvl*≥*n*.

If *side* = 'R', then *ldvl*≥ 1 .

ldvr INTEGER. The leading dimension of *vr*;

If *side* = 'R' or 'B', then *ldvr*≥*n*.

If *side* = 'L', then *ldvr*≥ 1.

mm INTEGER. The number of columns in the arrays *vl* and/or *vr* (*mm*≥*m*).

rwork REAL for ctgevc DOUBLE PRECISION for ztgevc. Workspace array, size at least max (1, 2**n*). Used in complex flavors only.

Output Parameters

vl On exit, if *side* = 'L' or 'B', *vl* contains:

if *howmny* = 'A', the matrix *Y* of left eigenvectors of (*S*,*P*);

if *howmny* = 'B', the matrix *Q***Y*;

if *howmny* = 'S', the left eigenvectors of (*S*,*P*) specified by *select*, stored consecutively in the columns of *vl*, in the same order as their eigenvalues.

For real flavors:

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

vr On exit, if *side* = 'R' or 'B', *vr* contains:

if *howmny* = 'A', the matrix *X* of right eigenvectors of (*S*,*P*);

if *howmny* = 'B', the matrix *Z***X*;

if *howmny* = 'S', the right eigenvectors of (*S*,*P*) specified by *select*, stored consecutively in the columns of *vr*, in the same order as their eigenvalues.

For real flavors:

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

m INTEGER. The number of columns in the arrays *vl* and/or *vr* actually used to store the eigenvectors.

If *howmny* = 'A' or 'B', *m* is set to *n*.

For real flavors:

Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.

For complex flavors:

Each selected eigenvector occupies one column.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*-th parameter had an illegal value.

For real flavors:

if *info* = *i*>0, the 2-by-2 block (*i*:*i*+1) does not have a complex eigenvalue.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine tgevc interface are the following:

s Holds the matrix *S* of size (*n*,*n*).

p Holds the matrix *P* of size (*n*,*n*).

select Holds the vector of length *n*.

vl Holds the matrix *VL* of size (*n*,*mm*).

vr Holds the matrix *VR* of size (*n*,*mm*).

side Restored based on the presence of arguments *vl* and *vr* as follows:

side = 'B', if both *vl* and *vr* are present,

side = 'L', if *vl* is present and *vr* omitted,

side = 'R', if *vl* is omitted and *vr* present,

Note that there will be an error condition if both *vl* and *vr* are omitted.

howmny If omitted, this argument is restored based on the presence of argument *select* as follows:

howmny = 'S', if *select* is present,

howmny = 'A', if *select* is omitted.

If present, *howmny* must be equal to 'A' or 'B' and the argument *select* must be omitted.

Note that there will be an error condition if both *howmny* and *select* are present.

?tgexc
Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.

Syntax

```
call stgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work, lwork, info)
call dtgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work, lwork, info)
call ctgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)
call ztgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)
call tgexc(a, b [,ifst] [,ilst] [,z] [,q] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (*A*,*B*) using an orthogonal/unitary equivalence transformation

$(A,B) = Q*(A,B)*Z^H$,

so that the diagonal block of (*A*, *B*) with row index *ifst* is moved to row *ilst*. Matrix pair (*A*, *B*) must be in a generalized real-Schur/Schur canonical form (as returned by gges), that is, *A* is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks and *B* is upper triangular. Optionally, the matrices *Q* and *Z* of generalized Schur vectors are updated.

$Q_{in}*A_{in}*Z_{in}^T = Q_{out}*A_{out}*Z_{out}^T$

$Q_{in}*B_{in}*Z_{in}^T = Q_{out}*B_{out}*Z_{out}^T$.

Input Parameters

wantq, *wantz*	LOGICAL. If *wantq* = .TRUE., update the left transformation matrix *Q*; If *wantq* = .FALSE., do not update *Q*; If *wantz* = .TRUE., update the right transformation matrix *Z*; If *wantz* = .FALSE., do not update *Z*.
n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
a, *b*, *q*, *z*	REAL for stgexc

`DOUBLE PRECISION` for `dtgexc`

`COMPLEX` for `ctgexc`

`DOUBLE COMPLEX` for `ztgexc`.

Arrays:

a(*lda*,*) contains the matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *n*).

q(*ldq*,*)

If *wantq* = .FALSE., then *q* is not referenced.

If *wantq* = .TRUE., then *q* must contain the orthogonal/unitary matrix *Q*.

The second dimension of *q* must be at least max(1, *n*).

z(*ldz*,*)

If *wantz* = .FALSE., then *z* is not referenced.

If *wantz* = .TRUE., then *z* must contain the orthogonal/unitary matrix *Z*.

The second dimension of *z* must be at least max(1, *n*).

lda — `INTEGER`. The leading dimension of *a*; at least max(1, *n*).

ldb — `INTEGER`. The leading dimension of *b*; at least max(1, *n*).

ldq — `INTEGER`. The leading dimension of *q*;

If *wantq* = .FALSE., then *ldq*≥ 1.

If *wantq* = .TRUE., then *ldq*≥ max(1, *n*).

ldz — `INTEGER`. The leading dimension of *z*;

If *wantz* = .FALSE., then *ldz*≥ 1.

If *wantz* = .TRUE., then *ldz*≥ max(1, *n*).

ifst, *ilst* — `INTEGER`. Specify the reordering of the diagonal blocks of (*A*, *B*). The block with row index *ifst* is moved to row *ilst*, by a sequence of swapping between adjacent blocks. Constraint: 1 ≤*ifst*, *ilst*≤*n*.

work — `REAL` for `stgexc`;

`DOUBLE PRECISION` for `dtgexc`.

Workspace array, size (*lwork*). Used in real flavors only.

lwork — `INTEGER`. The dimension of *work*; must be at least 4*n* +16.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

a, *b*, *q*, *z*	Overwritten by the updated matrices *A*,*B*, *Q*, and *Z* respectively.
ifst, *ilst*	Overwritten for real flavors only. If *ifst* pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; *ilst* always points to the first row of the block in its final position (which may differ from its input value by ±1).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is ill-conditioned. (*A*, *B*) may have been partially reordered, and *ilst* points to the first row of the current position of the block being moved.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine tgexc interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*n*).
z	Holds the matrix *Z* of size (*n*,*n*).
q	Holds the matrix *Q* of size (*n*,*n*).
wantq	Restored based on the presence of the argument *q* as follows: *wantq* = .TRUE., if *q* is present, *wantq* = .FALSE., if *q* is omitted.
wantz	Restored based on the presence of the argument *z* as follows: *wantz* = .TRUE., if *z* is present, *wantz* = .FALSE., if z is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsen
Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).

Syntax

`call stgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar, alphai, beta, q, ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)`

`call dtgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar, alphai, beta, q, ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)`

`call ctgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q, ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)`

`call ztgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q, ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)`

`call tgsen(a, b, select [,alphar] [,alphai] [,beta] [,ijob] [,q] [,z] [,pl] [,pr] [,dif] [,m] [,info])`

`call tgsen(a, b, select [,alpha] [,beta] [,ijob] [,q] [,z] [,pl] [,pr] [, dif] [,m] [,info])`

Include Files

- `mkl.fi, lapack.f90`

Description

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (*A*, *B*) (in terms of an orthogonal/unitary equivalence transformation $Q^T*(A,B)*Z$ for real flavors or $Q^H*(A,B)*Z$ for complex flavors), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (*A*, *B*). The leading columns of *Q* and *Z* form orthonormal/unitary bases of the corresponding left and right eigenspaces (deflating subspaces).

(*A*, *B*) must be in generalized real-Schur/Schur canonical form (as returned by gges), that is, *A* and *B* are both upper triangular.

`?tgsen` also computes the generalized eigenvalues

ω_j = (*alphar*(j) + *alphai*(j)*i)/*beta*(j) (for real flavors)

ω_j = *alpha*(j)/*beta*(j) (for complex flavors)

of the reordered matrix pair (*A*, *B*).

Optionally, the routine computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are `Difu[(`A_{11}`, `B_{11}`), (`A_{22}`, `B_{22}`)]` and `Difl[(`A_{11}`, `B_{11}`), (`A_{22}`, `B_{22}`)]`, that is, the separation(s) between the matrix pairs (A_{11}, B_{11}) and (A_{22}, B_{22}) that correspond to the selected cluster and the eigenvalues outside the cluster, respectively, and norms of "projections" onto left and right eigenspaces with respect to the selected cluster in the (1,1)-block.

Input Parameters

ijob — `INTEGER`. Specifies whether condition numbers are required for the cluster of eigenvalues (*pl* and *pr*) or the deflating subspaces `Difu` and `Difl`.

If *ijob* =0, only reorder with respect to *select*;

If *ijob* =1, reciprocal of norms of "projections" onto left and right eigenspaces with respect to the selected cluster (*pl* and *pr*);

If *ijob* =2, compute upper bounds on Difu and Difl, using F-norm-based estimate (*dif* (1:2));

If *ijob* =3, compute estimate of Difu and Difl, using 1-norm-based estimate (*dif* (1:2)). This option is about 5 times as expensive as *ijob* =2;

If *ijob* =4,>compute *pl*, *pr* and *dif* (i.e., options 0, 1 and 2 above). This is an economic version to get it all;

If *ijob* =5, compute *pl*, *pr* and *dif* (i.e., options 0, 1 and 3 above).

wantq, *wantz*

LOGICAL.

If *wantq* = .TRUE., update the left transformation matrix *Q*;

If *wantq* = .FALSE., do not update *Q*;

If *wantz* = .TRUE., update the right transformation matrix *Z*;

If *wantz* = .FALSE., do not update *Z*.

select

LOGICAL.

Array, size at least max (1, *n*). Specifies the eigenvalues in the selected cluster.

To select an eigenvalue ω_j, *select*(*j*) must be .TRUE.*For real flavors*: to select a complex conjugate pair of eigenvalues ω_j and ω_{j+1} (corresponding 2 by 2 diagonal block), *select*(*j*) and/or *select*(*j* + 1) must be set to .TRUE.; the complex conjugate ω_j and ω_{j+1} must be either both included in the cluster or both excluded.

n

INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, *b*, *q*, *z*, *work*

REAL for stgsen

DOUBLE PRECISION for dtgsen

COMPLEX for ctgsen

DOUBLE COMPLEX for ztgsen.

Arrays:

a(*lda*,*) contains the matrix *A*.

For real flavors: *A* is upper quasi-triangular, with (*A*, *B*) in generalized real Schur canonical form.

For complex flavors: *A* is upper triangular, in generalized Schur canonical form.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the matrix *B*.

For real flavors: *B* is upper triangular, with (*A*, *B*) in generalized real Schur canonical form.

For complex flavors: *B* is upper triangular, in generalized Schur canonical form. The second dimension of *b* must be at least max(1, *n*).

q(*ldq*,*)

If *wantq* = .TRUE., then *q* is an *n*-by-*n* matrix;

If *wantq* = .FALSE., then *q* is not referenced.

The second dimension of *q* must be at least max(1, *n*).

z(*ldz*,*)

If *wantz* = .TRUE., then *z* is an *n*-by-*n* matrix;

If *wantz* = .FALSE., then *z* is not referenced.

The second dimension of *z* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldb INTEGER. The leading dimension of *b*; at least max(1, *n*).

ldq INTEGER. The leading dimension of *q*; *ldq*≥ 1.

If *wantq* = .TRUE., then *ldq*≥ max(1, *n*).

ldz INTEGER. The leading dimension of *z*; *ldz*≥ 1.

If *wantz* = .TRUE., then *ldz*≥ max(1, *n*).

lwork INTEGER. The dimension of the array *work*.

For real flavors:

If *ijob* = 1, 2, or 4, *lwork*≥ max(4*n*+16, 2*m*(*n*-*m*)).

If *ijob* = 3 or 5, *lwork*≥ max(4*n*+16, 4*m*(*n*-*m*)).

For complex flavors:

If *ijob* = 1, 2, or 4, *lwork*≥ max(1, 2*m*(*n*-*m*)).

If *ijob* = 3 or 5, *lwork*≥ max(1, 4*m*(*n*-*m*)).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER. The dimension of the array *iwork*.

For real flavors:

If *ijob* = 1, 2, or 4, *liwork*≥*n*+6.

If *ijob* = 3 or 5, *liwork*≥ max(*n*+6, 2*m*(*n*-*m*)).

For complex flavors:

If *ijob* = 1, 2, or 4, *liwork*≥*n*+2.

If *ijob* = 3 or 5, *liwork*≥ max(*n*+2, 2*m*(*n*-*m*)).

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

a, *b*

Overwritten by the reordered matrices *A* and *B*, respectively.

alphar, *alphai*

`REAL` for `stgsen`;

`DOUBLE PRECISION` for `dtgsen`.

Arrays, size at least max(1, *n*). Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha

`COMPLEX` for `ctgsen`;

`DOUBLE COMPLEX` for `ztgsen`.

Array, size at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors.

See *beta*.

beta

`REAL` for `stgsen`

`DOUBLE PRECISION` for `dtgsen`

`COMPLEX` for `ctgsen`

`DOUBLE COMPLEX` for `ztgsen`.

Array, size at least max(1, *n*).

For real flavors:

On exit, `(alphar(j) + alphai(j)*i)/beta(j)`, *j*=1,..., *n*, will be the generalized eigenvalues.

`alphar(j) + alphai(j)*i` and `beta(j)`, *j*=1,..., *n* are the diagonals of the complex Schur form (*S*,*T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A*,*B*) were further reduced to triangular form using complex unitary transformations.

If `alphai(j)` is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j* + 1)-st eigenvalues are a complex conjugate pair, with `alphai(j + 1)` negative.

For complex flavors:

The diagonal elements of *A* and *B*, respectively, when the pair (*A*,*B*) has been reduced to generalized Schur form. `alpha(i)/beta(i)`,*i*=1,..., *n* are the generalized eigenvalues.

q

If *wantq* = `.TRUE.`, then, on exit, *Q* has been postmultiplied by the left orthogonal transformation matrix which reorder (*A*, *B*). The leading *m* columns of *Q* form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

z

If *wantz* = `.TRUE.`, then, on exit, *Z* has been postmultiplied by the left orthogonal transformation matrix which reorder (*A*, *B*). The leading *m* columns of *Z* form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

m

`INTEGER`.

The dimension of the specified pair of left and right eigen-spaces (deflating subspaces); $0 \le m \le n$.

pl, *pr*

`REAL` for single precision flavors;

`DOUBLE PRECISION` for double precision flavors.

If *ijob* = 1, 4, or 5, *pl* and *pr* are lower bounds on the reciprocal of the norm of "projections" onto left and right eigenspaces with respect to the selected cluster.

$0 < pl, pr \le 1$. If $m = 0$ or $m = n$, $pl = pr = 1$.

If *ijob* = 0, 2 or 3, *pl* and *pr* are not referenced

dif

`REAL` for single precision flavors;`DOUBLE PRECISION` for double precision flavors.

Array, size (2).

If $ijob \ge 2$, *dif*(1:2) store the estimates of `Difu` and `Difl`.

If *ijob* = 2 or 4, *dif*(1:2) are F-norm-based upper bounds on `Difu` and `Difl`.

If *ijob* = 3 or 5, *dif*(1:2) are 1-norm-based estimates of `Difu` and `Difl`.

If $m = 0$ or $m = n$, *dif*(1:2) = F-norm([*A*, *B*]).

If *ijob* = 0 or 1, *dif* is not referenced.

work(1)

If *ijob* is not 0 and *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

iwork(1)

If *ijob* is not 0 and *info* = 0, on exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance. Use this *liwork* for subsequent runs.

info

`INTEGER`.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, Reordering of (*A*, *B*) failed because the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is very ill-conditioned. (*A*, *B*) may have been partially reordered.

If *ijob* > 0, 0 is returned in *dif*, *pl* and *pr*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `tgsen` interface are the following:

a

Holds the matrix *A* of size (*n*,*n*).

b

Holds the matrix *B* of size (*n*,*n*).

select

Holds the vector of length *n*.

alphar	Holds the vector of length *n*. Used in real flavors only.
alphai	Holds the vector of length *n*. Used in real flavors only.
alpha	Holds the vector of length *n*. Used in complex flavors only.
beta	Holds the vector of length *n*.
q	Holds the matrix *Q* of size (*n*,*n*).
z	Holds the matrix *Z* of size (*n*,*n*).
dif	Holds the vector of length (2).
ijob	Must be 0, 1, 2, 3, 4, or 5. The default value is 0.
wantq	Restored based on the presence of the argument *q* as follows: *wantq* = .TRUE., if *q* is present, *wantq* = .FALSE., if *q* is omitted.
wantz	Restored based on the presence of the argument *z* as follows: *wantz* = .TRUE., if *z* is present, *wantz* = .FALSE., if z is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsyl
Solves the generalized Sylvester equation.

Syntax

```
call stgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale,
dif, work, lwork, iwork, info)

call dtgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale,
dif, work, lwork, iwork, info)

call ctgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale,
dif, work, lwork, iwork, info)

call ztgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale,
dif, work, lwork, iwork, info)

call tgsyl(a, b, c, d, e, f [,ijob] [,trans] [,scale] [,dif] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves the generalized Sylvester equation:

*A*R-L*B = scale*C*

*D*R-L*E = scale*F*

where *R* and *L* are unknown *m*-by-*n* matrices, (*A*, *D*), (*B*, *E*) and (*C*, *F*) are given matrix pairs of size *m*-by-*m*, *n*-by-*n* and *m*-by-*n*, respectively, with real/complex entries. (*A*, *D*) and (*B*, *E*) must be in generalized real-Schur/Schur canonical form, that is, *A*, *B* are upper quasi-triangular/triangular and *D*, *E* are upper triangular.

The solution (*R*, *L*) overwrites (*C*, *F*). The factor *scale*, $0 \le scale \le 1$, is an output scaling factor chosen to avoid overflow.

In matrix notation the above equation is equivalent to the following: solve *Z*x = scale*b*, where *Z* is defined as

$$Z = \begin{pmatrix} kron(I_n, A) & -kron(B^T, I_m) \\ kron(I_n, D) & -kron(E^T, I_m) \end{pmatrix}$$

Here I_k is the identity matrix of size *k* and X^T is the transpose/conjugate-transpose of *X*. *kron(X, Y)* is the Kronecker product between the matrices *X* and *Y*.

If *trans = 'T'* (for real flavors), or *trans = 'C'* (for complex flavors), the routine ?tgsyl solves the transposed/conjugate-transposed system $Z^T*y = scale*b$, which is equivalent to solve for *R* and *L* in

$A^T*R+D^T*L = scale*C$

$R*B^T+L*E^T = scale*(-F)$

This case (*trans = 'T'* for stgsyl/dtgsyl or *trans = 'C'* for ctgsyl/ztgsyl) is used to compute an one-norm-based estimate of Dif[(*A*, *D*), (*B*, *E*)], the separation between the matrix pairs (*A*,*D*) and (*B*,*E*), using lacon/lacon.

If *ijob* ≥ 1, ?tgsyl computes a Frobenius norm-based estimate of Dif[(*A*, *D*), (*B*,*E*)]. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of *Z*. This is a level 3 BLAS algorithm.

Input Parameters

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.

If *trans = 'N'*, solve the generalized Sylvester equation.

If *trans = 'T'*, solve the 'transposed' system (for real flavors only).

If *trans = 'C'*, solve the ' conjugate transposed' system (for complex flavors only).

ijob

INTEGER. Specifies what kind of functionality to be performed:

If *ijob* =0, solve the generalized Sylvester equation only;

If *ijob* =1, perform the functionality of *ijob* =0 and *ijob* =3;

If *ijob* =2, perform the functionality of *ijob* =0 and *ijob* =4;

If *ijob* =3, only an estimate of Dif[(*A*, *D*), (*B*, *E*)] is computed (look ahead strategy is used);

If *ijob* =4, only an estimate of Dif[(*A*, *D*), (*B*,*E*)] is computed (?gecon on sub-systems is used). If *trans* = 'T' or 'C', *ijob* is not referenced.

m — INTEGER. The order of the matrices *A* and *D*, and the row dimension of the matrices *C*, *F*, *R* and *L*.

n — INTEGER. The order of the matrices *B* and *E*, and the column dimension of the matrices *C*, *F*, *R* and *L*.

a, *b*, *c*, *d*, *e*, *f*, *work* — REAL for stgsyl

DOUBLE PRECISION for dtgsyl

COMPLEX for ctgsyl

DOUBLE COMPLEX for ztgsyl.

Arrays:

a(*lda*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *A*.

The second dimension of *a* must be at least max(1, *m*).

b(*ldb*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *B*. The second dimension of *b* must be at least max(1, *n*).

c(*ldc*,*) contains the right-hand-side of the first matrix equation in the generalized Sylvester equation (as defined by *trans*)

The second dimension of *c* must be at least max(1, *n*).

d(*ldd*,*) contains the upper triangular matrix *D*.

The second dimension of *d* must be at least max(1, *m*).

e(*lde*,*) contains the upper triangular matrix *E*.

The second dimension of *e* must be at least max(1, *n*).

f(*ldf*,*) contains the right-hand-side of the second matrix equation in the generalized Sylvester equation (as defined by *trans*)

The second dimension of *f* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; at least max(1, *m*).

ldb — INTEGER. The leading dimension of *b*; at least max(1, *n*).

ldc — INTEGER. The leading dimension of *c*; at least max(1, *m*) .

ldd — INTEGER. The leading dimension of *d*; at least max(1, *m*).

lde — INTEGER. The leading dimension of *e*; at least max(1, *n*).

ldf — INTEGER. The leading dimension of *f*; at least max(1, *m*) .

lwork INTEGER.

The dimension of the array *work*. *lwork* ≥ 1.

If *ijob* = 1 or 2 and *trans* = 'N', *lwork* ≥ max(1, 2**m***n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for details.

iwork INTEGER. Workspace array, size at least (*m*+*n*+6) for real flavors, and at least (*m*+*n*+2) for complex flavors.

Output Parameters

c If *ijob*=0, 1, or 2, overwritten by the solution *R*.

If *ijob*=3 or 4 and *trans* = 'N', *c* holds *R*, the solution achieved during the computation of the Dif-estimate.

f If *ijob*=0, 1, or 2, overwritten by the solution *L*.

If *ijob*=3 or 4 and *trans* = 'N', *f* holds *L*, the solution achieved during the computation of the Dif-estimate.

dif REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

On exit, *dif* is the reciprocal of a lower bound of the reciprocal of the Dif-function, that is, *dif* is an upper bound of Dif[(*A*, *D*), (*B*, *E*)] = sigma_min(*Z*), where *Z* as defined in the description.

If *ijob* = 0, or *trans* = 'T' (for real flavors), or *trans* = 'C' (for complex flavors), *dif* is not touched.

scale REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

On exit, *scale* is the scaling factor in the generalized Sylvester equation.

If 0 < *scale* < 1, *c* and *f* hold the solutions *R* and *L*, respectively, to a slightly perturbed system but the input matrices *A*, *B*, *D* and *E* have not been changed.

If *scale* = 0, *c* and *f* hold the solutions *R* and *L*, respectively, to the homogeneous system with *C* = *F* = 0. Normally, *scale* = 1.

work(1) If *info* = 0, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, (*A*, *D*) and (*B*, *E*) have common or close eigenvalues.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `tgsyl` interface are the following:

a	Holds the matrix *A* of size (*m*,*m*).
b	Holds the matrix *B* of size (*n*,*n*).
c	Holds the matrix *C* of size (*m*,*n*).
d	Holds the matrix *D* of size (*m*,*m*).
e	Holds the matrix *E* of size (*n*,*n*).
f	Holds the matrix *F* of size (*m*,*n*).
ijob	Must be 0, 1, 2, 3, or 4. The default value is 0.
trans	Must be `'N'` or `'T'`. The default value is `'N'`.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsna

Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

Syntax

```
call stgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm, m,
work, lwork, iwork, info)
```

```
call dtgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm, m,
work, lwork, iwork, info)
```

```
call ctgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm, m,
work, lwork, iwork, info)
```

```
call ztgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm, m,
work, lwork, iwork, info)
```

```
call tgsna(a, b [,s] [,dif] [,vl] [,vr] [,select] [,m] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The real flavors stgsna/dtgsna of this routine estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (*A*, *B*) in generalized real Schur canonical form (or of any matrix pair ($Q*A*Z^T$, $Q*B*Z^T$) with orthogonal matrices *Q* and *Z*.

(*A*, *B*) must be in generalized real Schur form (as returned by gges/gges), that is, *A* is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex flavors ctgsna/ztgsna estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (*A*, *B*). (*A*, *B*) must be in generalized Schur canonical form, that is, *A* and *B* are both upper triangular.

Input Parameters

job

CHARACTER*1. Specifies whether condition numbers are required for eigenvalues or eigenvectors. Must be 'E' or 'V' or 'B'.

If *job* = 'E', for eigenvalues only (compute *s*).

If *job* = 'V', for eigenvectors only (compute *dif*).

If *job* = 'B', for both eigenvalues and eigenvectors (compute both *s* and *dif*).

howmny

CHARACTER*1. Must be 'A' or 'S'.

If *howmny* = 'A', compute condition numbers for all eigenpairs.

If *howmny* = 'S', compute condition numbers for selected eigenpairs specified by the logical array *select*.

select

LOGICAL.

Array, size at least max (1, *n*).

If *howmny* = 'S', *select* specifies the eigenpairs for which condition numbers are required.

If *howmny* = 'A', *select* is not referenced.

For real flavors:

To select condition numbers for the eigenpair corresponding to a real eigenvalue ω_j, *select*(*j*) must be set to .TRUE.; to select condition numbers corresponding to a complex conjugate pair of eigenvalues ω_j and ω_{j+1}, either *select*(*j*) or *select*(*j* + 1) must be set to .TRUE.

For complex flavors:

To select condition numbers for the corresponding *j*-th eigenvalue and/or eigenvector, *select*(*j*) must be set to .TRUE..

n

INTEGER. The order of the square matrix pair (*A*, *B*)

($n \geq 0$).

a, *b*, *vl*, *vr*, *work*

REAL for stgsna

`DOUBLE PRECISION` for `dtgsna`

`COMPLEX` for `ctgsna`

`DOUBLE COMPLEX` for `ztgsna`.

Arrays:

a(*lda*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *A* in the pair (*A*, *B*).

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the upper triangular matrix *B* in the pair (*A*, *B*). The second dimension of *b* must be at least max(1, *n*).

If *job* = `'E'` or `'B'`, *vl*(*ldvl*,*) must contain left eigenvectors of (*A*, *B*), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by `?tgevc`.

If *job* = `'V'`, *vl* is not referenced. The second dimension of *vl* must be at least max(1, *m*).

If *job* = `'E'` or `'B'`, *vr*(*ldvr*,*) must contain right eigenvectors of (*A*, *B*), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by `?tgevc`.

If *job* = `'V'`, *vr* is not referenced. The second dimension of *vr* must be at least max(1, *m*).

work is a workspace array, its dimension `max(1, lwork)`.

If *job* = `'E'`, *work* is not referenced.

lda — `INTEGER`. The leading dimension of *a*; at least max(1, *n*).

ldb — `INTEGER`. The leading dimension of *b*; at least max(1, *n*).

ldvl — `INTEGER`. The leading dimension of *vl*; *ldvl*≥ 1.

If *job* = `'E'` or `'B'`, then *ldvl*≥ max(1, *n*) .

ldvr — `INTEGER`. The leading dimension of *vr*; *ldvr*≥ 1.

If *job* = `'E'` or `'B'`, then *ldvr*≥ `max(1, n)`.

mm — `INTEGER`. The number of elements in the arrays *s* and *dif* (*mm*≥*m*).

lwork — `INTEGER`. The dimension of the array *work*.

lwork≥ `max(1, n)`.

If *job* = `'V'` or `'B'`, *lwork*≥ `2*n*(n+2)+16` for real flavors, and *lwork*≥ `max(1, 2*n*n)` for complex flavors.

If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for details.

iwork — `INTEGER`. Workspace array, size at least (*n*+6) for real flavors, and at least (*n*+2) for complex flavors.

If *job* = 'E', *iwork* is not referenced.

Output Parameters

s

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size (*mm*).

If *job* = 'E' or 'B', contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array.

If *job* = 'V', *s* is not referenced.

For real flavors:

For a complex conjugate pair of eigenvalues two consecutive elements of *s* are set to the same value. Thus, *s*(*j*), *dif*(*j*), and the *j*-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the *j*-th eigenpair, unless all eigenpairs are selected).

dif

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size (*mm*).

If *job* = 'V' or 'B', contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array.

If the eigenvalues cannot be reordered to compute *dif*(*j*), *dif*(*j*) is set to 0; this can only occur when the true value would be very small anyway.

If *job* = 'E', *dif* is not referenced.

For real flavors:

For a complex eigenvector, two consecutive elements of *dif* are set to the same value.

For complex flavors:

For each eigenvalue/vector specified by *select*, *dif* stores a Frobenius norm-based estimate of Difl.

m

INTEGER. The number of elements in the arrays *s* and *dif* used to store the specified condition numbers; for each selected eigenvalue one element is used.

If *howmny* = 'A', *m* is set to *n*.

work(1)

work(1)

If *job* is not 'E' and *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `tgsna` interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*n*).
s	Holds the vector of length (*mm*).
dif	Holds the vector of length (*mm*).
vl	Holds the matrix *VL* of size (*n*,*mm*).
vr	Holds the matrix *VR* of size (*n*,*mm*).
select	Holds the vector of length *n*.
howmny	Restored based on the presence of the argument *select* as follows: *howmny* = 'S', if *select* is present, *howmny* = 'A', if *select* is omitted.
job	Restored based on the presence of arguments *s* and dif as follows: *job* = 'B', if both *s* and *dif* are present, *job* = 'E', if *s* is present and *dif* omitted, *job* = 'V', if *s* is omitted and *dif* present, Note that there will be an error condition if both *s* and *dif* are omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Generalized Singular Value Decomposition: LAPACK Computational Routines

This topic describes LAPACK computational routines used for finding the generalized singular value decomposition (GSVD) of two matrices *A* and *B* as

$U^H A Q = D_1 * (0 \ R)$,

$V^H B Q = D_2 * (0 \ R)$,

where *U*, *V*, and *Q* are orthogonal/unitary matrices, *R* is a nonsingular upper triangular matrix, and D_1, D_2 are "diagonal" matrices of the structure detailed in the routines description section.

Table "Computational Routines for Generalized Singular Value Decomposition" lists LAPACK routines (FORTRAN 77 interface) that perform generalized singular value decomposition of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Generalized Singular Value Decomposition

Routine name	Operation performed
ggsvp	Computes the preprocessing decomposition for the generalized SVD
ggsvp3	Performs preprocessing for a generalized SVD.
ggsvd3	Computes generalized SVD.
tgsja	Computes the generalized SVD of two upper triangular or trapezoidal matrices

You can use routines listed in the above table as well as the driver routine ggsvd to find the GSVD of a pair of general rectangular matrices.

?ggsvp
Computes the preprocessing decomposition for the generalized SVD (deprecated).

Syntax

```
call sggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v, ldv, q, ldq, iwork, tau, work, info)
call dggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v, ldv, q, ldq, iwork, tau, work, info)
call cggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info)
call zggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info)
call ggsvp(a, b, tola, tolb [, k] [,l] [,u] [,v] [,q] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

This routine is deprecated; use ggsvp3.

The routine computes orthogonal matrices *U*, *V* and *Q* such that

$$U^H A Q = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} \end{array}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix} \end{array}, \quad \text{if } m-k-l < 0$$

$$V^H B Q = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

where the k-by-k matrix A_{12} and l-by-l matrix B_{13} are nonsingular upper triangular; A_{23} is l-by-l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$-by-l upper trapezoidal. The sum $k+l$ is equal to the effective numerical rank of the $(m+p)$-by-n matrix $(A^H, B^H)^H$.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine ?tgsja.

Input Parameters

jobu

CHARACTER*1. Must be 'U' or 'N'.

If *jobu* = 'U', orthogonal/unitary matrix *U* is computed.

If *jobu* = 'N', *U* is not computed.

jobv

CHARACTER*1. Must be 'V' or 'N'.

If *jobv* = 'V', orthogonal/unitary matrix *V* is computed.

If *jobv* = 'N', *V* is not computed.

jobq

CHARACTER*1. Must be 'Q' or 'N'.

If *jobq* = 'Q', orthogonal/unitary matrix *Q* is computed.

If *jobq* = 'N', *Q* is not computed.

m

INTEGER. The number of rows of the matrix *A* ($m \geq 0$).

p

INTEGER. The number of rows of the matrix *B* ($p \geq 0$).

n

INTEGER. The number of columns of the matrices *A* and *B* ($n \geq 0$).

a, *b*, *tau*, *work*

REAL for sggsvp

DOUBLE PRECISION for dggsvp

COMPLEX for cggsvp

DOUBLE COMPLEX for zggsvp.

Arrays:

a(*lda*,*) contains the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the *p*-by-*n* matrix *B*.

The second dimension of *b* must be at least max(1, *n*).

tau(*) is a workspace array.

The dimension of *tau* must be at least max(1, *n*).

work(*) is a workspace array.

The dimension of *work* must be at least max(1, 3*n*, *m*, *p*).

lda INTEGER. The leading dimension of *a*; at least max(1, *m*).

ldb INTEGER. The leading dimension of *b*; at least max(1, *p*).

tola, *tolb* REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

tola and *tolb* are the thresholds to determine the effective numerical rank of matrix *B* and a subblock of *A*. Generally, they are set to

tola = max(*m*, *n*)*||*A*||*MACHEPS,

tolb = max(*p*, *n*)*||*B*||*MACHEPS.

The size of *tola* and *tolb* may affect the size of backward errors of the decomposition.

ldu INTEGER. The leading dimension of the output array *u* . *ldu*≥ max(1, *m*) if *jobu* = 'U'; *ldu*≥ 1 otherwise.

ldv INTEGER. The leading dimension of the output array *v* . *ldv*≥ max(1, *p*) if *jobv* = 'V'; *ldv*≥ 1 otherwise.

ldq INTEGER. The leading dimension of the output array *q* . *ldq*≥ max(1, *n*) if *jobq* = 'Q'; *ldq*≥ 1 otherwise.

iwork INTEGER. Workspace array, size at least max(1, *n*).

rwork REAL for cggsvp

DOUBLE PRECISION for zggsvp.

Workspace array, size at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

a Overwritten by the triangular (or trapezoidal) matrix described in the *Description* section.

b Overwritten by the triangular matrix described in the *Description* section.

k, *l* INTEGER. On exit, *k* and *l* specify the dimension of subblocks. The sum *k* + *l* is equal to effective numerical rank of $(A^H, B^H)^H$.

u, *v*, *q* REAL for sggsvp

DOUBLE PRECISION for dggsvp

COMPLEX for cggsvp

DOUBLE COMPLEX for zggsvp.

Arrays:

If *jobu* = 'U', *u*(*ldu*,*) contains the orthogonal/unitary matrix *U*.

The second dimension of *u* must be at least max(1, *m*).

If *jobu* = 'N', *u* is not referenced.

If *jobv* = 'V', *v*(*ldv*,*) contains the orthogonal/unitary matrix *V*.

The second dimension of *v* must be at least max(1, *m*).

If *jobv* = 'N', *v* is not referenced.

If *jobq* = 'Q', *q*(*ldq*,*) contains the orthogonal/unitary matrix *Q*.

The second dimension of *q* must be at least max(1, *n*).

If *jobq* = 'N', *q* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ggsvp interface are the following:

a Holds the matrix *A* of size (*m*,*n*).

b Holds the matrix *B* of size (*p*,*n*).

u Holds the matrix *U* of size (*m*,*m*).

v Holds the matrix *V* of size (*p*,*m*).

q Holds the matrix *Q* of size (*n*,*n*).

jobu Restored based on the presence of the argument *u* as follows:

jobu = 'U', if *u* is present,

jobu = 'N', if *u* is omitted.

jobv Restored based on the presence of the argument *v* as follows:

jobz = 'V', if *v* is present,

jobz = 'N', if *v* is omitted.

jobq Restored based on the presence of the argument *q* as follows:

jobz = 'Q', if *q* is present,

jobz = 'N', if *q* is omitted.

?ggsvp3

Performs preprocessing for a generalized SVD.

Syntax

```
call sggsvp3 (jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, tau, work, lwork, info )
```

```
call dggsvp3 (jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, tau, work, lwork, info )
```

```
call cggsvp3 (jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, rwork, tau, work, lwork, info )
```

```
call zggsvp3 (jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, rwork, tau, work, lwork, info )
```

Include Files

- mkl_lapack.fi

Include Files

- mkl.fi

Description

?ggsvp3 computes orthogonal or unitary matrices *U*, *V*, and *Q* such that

for real flavors:

$$U^T A Q = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A12 & A13 \\ 0 & 0 & A23 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$ if $m - k - l \geq 0$;

$$U^T A Q = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A12 & A13 \\ 0 & 0 & A23 \end{pmatrix} \end{array}$$ if $m - k - l < 0$;

$$V^T B Q = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B13 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

for complex flavors:

$$U^H A Q = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A12 & A13 \\ 0 & 0 & A23 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$ if $m - k - l \geq 0$;

$$U^H A Q = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A12 & A13 \\ 0 & 0 & A23 \end{pmatrix} \end{array}$$ if $m - k - l < 0$;

$$V^H B Q = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B13 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

where the k-by-k matrix *A12* and l-by-l matrix *B13* are nonsingular upper triangular; *A23* is l-by-l upper triangular if $m-k-l \geq 0$, otherwise *A23* is ($m-k$-by-l upper trapezoidal. $k + l$ = the effective numerical rank of the ($m + p$)-by-n matrix $(A^T,B^T)^T$ for real flavors or $(A^H,B^H)^H$ for complex flavors.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see ?ggsvd3.

Input Parameters

jobu

CHARACTER*1. = 'U': Orthogonal/unitary matrix *U* is computed;

= 'N': *U* is not computed.

jobv CHARACTER*1. = 'V': Orthogonal/unitary matrix *V* is computed;
= 'N': *V* is not computed.

jobq CHARACTER*1. = 'Q': Orthogonal/unitary matrix *Q* is computed;
= 'N': *Q* is not computed.

m INTEGER. The number of rows of the matrix *A*.
m≥ 0.

p INTEGER. The number of rows of the matrix *B*.
p≥ 0.

n INTEGER. The number of columns of the matrices *A* and B.
n≥ 0.

a `REAL` for `sggsvp3`
`DOUBLE PRECISION` for `dggsvp3`
`COMPLEX` for `cggsvp3`
`DOUBLE COMPLEX` for `zggsvp3`
Array, size (*lda*, *n*).
On entry, the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.
lda≥ max(1,*m*).

b `REAL` for `sggsvp3`
`DOUBLE PRECISION` for `dggsvp3`
`COMPLEX` for `cggsvp3`
`DOUBLE COMPLEX` for `zggsvp3`
Array, size (*ldb*, *n*).
On entry, the *p*-by-*n* matrix *B*.

ldb INTEGER. The leading dimension of the array *b*.
ldb≥ max(1,*p*).

tola, *tolb* `REAL` for `sggsvp3`
`DOUBLE PRECISION` for `dggsvp3`
`REAL` for `cggsvp3`
`DOUBLE PRECISION` for `zggsvp3`
tola and *tolb* are the thresholds to determine the effective numerical rank of matrix *B* and a subblock of *A*. Generally, they are set to
tola = max(*m*,*n*)*norm(*a*)**MACHEPS*,
tolb = max(*p*,*n*)*norm(*b*)**MACHEPS*.

The size of *tola* and *tolb* may affect the size of backward errors of the decomposition.

ldu INTEGER. The leading dimension of the array *u*.

ldu≥ max(1,*m*) if *jobu* = 'U'; *ldu*≥ 1 otherwise.

ldv INTEGER. The leading dimension of the array *v*.

ldv≥ max(1,*p*) if *jobv* = 'V'; *ldv*≥ 1 otherwise.

ldq INTEGER. The leading dimension of the array *q*.

ldq≥ max(1,*n*) if *jobq* = 'Q'; *ldq*≥ 1 otherwise.

iwork INTEGER. Array, size (*n*).

rwork for `sggsvp3`

for `dggsvp3`

`REAL` for `cggsvp3`

`DOUBLE PRECISION` for `zggsvp3`

Array, size (2**n*).

tau `REAL` for `sggsvp3`

`DOUBLE PRECISION` for `dggsvp3`

`COMPLEX` for `cggsvp3`

`DOUBLE COMPLEX` for `zggsvp3`

Array, size (*n*).

The scalar factors of the elementary reflectors.

work `REAL` for `sggsvp3`

`DOUBLE PRECISION` for `dggsvp3`

`COMPLEX` for `cggsvp3`

`DOUBLE COMPLEX` for `zggsvp3`

Array, size (MAX(1,*lwork*)).

lwork INTEGER. The dimension of the array *work*.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.

Output Parameters

a On exit, *a* contains the triangular (or trapezoidal) matrix described in the Description section.

b On exit, *b* contains the triangular matrix described in the Description section.

k, *l*	INTEGER. On exit, *k* and *l* specify the dimension of the subblocks described in Description section. *k* + *l* = effective numerical rank of $(A^T,B^T)^T$ for real flavors or $(A^H,B^H)^H$ for complex flavors.
u	REAL for sggsvp3 DOUBLE PRECISION for dggsvp3 COMPLEX for cggsvp3 DOUBLE COMPLEX for zggsvp3 Array, size (*ldu*, *m*). If *jobu* = 'U', *u* contains the orthogonal/unitary matrix *U*. If *jobu* = 'N', *u* is not referenced.
v	REAL for sggsvp3 DOUBLE PRECISION for dggsvp3 COMPLEX for cggsvp3 DOUBLE COMPLEX for zggsvp3 Array, size (*ldv*, *p*). If *jobv* = 'V', *v* contains the orthogonal/unitary matrix *V*. If *jobv* = 'N', *v* is not referenced.
q	REAL for sggsvp3 DOUBLE PRECISION for dggsvp3 COMPLEX for cggsvp3 DOUBLE COMPLEX for zggsvp3 Array, size (*ldq*, *n*). If *jobq* = 'Q', *q* contains the orthogonal/unitary matrix *Q*. If *jobq* = 'N', *q* is not referenced.
work	On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.
info	INTEGER. = 0: successful exit. < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

Application Notes

The subroutine uses LAPACK subroutine ?geqp3 for the QR factorization with column pivoting to detect the effective numerical rank of the *A* matrix. It may be replaced by a better rank determination strategy.

?ggsvp3 replaces the deprecated subroutine ?ggsvp.

?ggsvd3
Computes generalized SVD.

Syntax

```
call sggsvd3(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, lwork, iwork, info)
```

```
call dggsvd3(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, lwork, iwork, info)
```

```
call cggsvd3(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, lwork, rwork, iwork, info)
```

```
call zggsvd3(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, lwork, rwork, iwork, info)
```

Include Files

- mkl.fi

Description

?ggsvd3 computes the generalized singular value decomposition (GSVD) of an m-by-n real or complex matrix *A* and p-by-n real or complex matrix *B*:

$U^T*A*Q = D_1*(0\ R\ ), V^T*B*Q = D_2*(0\ R\ )$ for real flavors

or

$U^H*A*Q = D_1*(0\ R\ ), V^H*B*Q = D_2*(0\ R\ )$ for complex flavors

where *U*, *V* and *Q* are orthogonal/unitary matrices.

Let $k+l$ = the effective numerical rank of the matrix $(A^TB^T)^T$ for real flavors or the matrix $(A^H,B^H)^H$ for complex flavors, then *R* is a $(k + l)$-by-$(k + l)$ nonsingular upper triangular matrix, D_1 and D_2 are m-by-$(k + l)$ and p-by-$(k + l)$ "diagonal" matrices and of the following structures, respectively:

If $m-k-l \geq 0$,

$$D_1 = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{array}$$

$$\begin{pmatrix} 0 & R \end{pmatrix} = \begin{array}{c} \\ k \\ l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & R11 & R12 \\ 0 & 0 & R22 \end{pmatrix} \end{array}$$

where

C = diag($alpha(k+1)$, ... , $alpha(k+l)$),

S = diag($beta(k+1)$, ... , $beta(k+l)$),

$C^2 + S^2 = I$.

If $m - k - l < 0$,

$$D_1 = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ m-k \\ k+l-m \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\begin{pmatrix} 0 & R \end{pmatrix} = \begin{array}{c} \\ k \\ m-k \\ k+l-m \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & R11 & R12 & R13 \\ 0 & 0 & R22 & R23 \\ 0 & 0 & 0 & R33 \end{pmatrix} \end{array}$$

where

C = diag(*alpha*(*k* + 1), ... , *alpha*(*m*)),

S = diag(*beta*(*k* + 1), ... , *beta*(*m*)),

$C^2 + S^2 = I$.

The routine computes *C*, *S*, *R*, and optionally the orthogonal/unitary transformation matrices *U*, *V* and *Q*.

In particular, if *B* is an n-by-n nonsingular matrix, then the GSVD of *A* and *B* implicitly gives the SVD of *A**inv(*B*):

$A*\text{inv}(B) = U*(D_1*\text{inv}(D_2))*V^T$ for real flavors

or

$A*\text{inv}(B) = U*(D_1*\text{inv}(D_2))*V^H$ for complex flavors.

If $(A^T,B^T)^T$ for real flavors or $(A^H,B^H)^H$ for complex flavors has orthonormal columns, then the GSVD of *A* and *B* is also equal to the CS decomposition of *A* and *B*. Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$A^T*AX = \lambda* B^T*BX$ for real flavors

or

$A^H*AX = \lambda* B^H*BX$ for complex flavors

In some literature, the GSVD of *A* and *B* is presented in the form

$U^T*A*X = (0\ D_1)$, $V^T*B*X = (0\ D_2)$ for real (*A*, *B*)

or

$U^H*A*X = (0\ D_1)$, $V^H*B*X = (0\ D_2)$ for complex (*A*, *B*)

where *U* and *V* are orthogonal and *X* is nonsingular, D_1 and D_2 are "diagonal''. The former GSVD form can be converted to the latter form by taking the nonsingular matrix *X* as

$$X = Q*\begin{pmatrix} I & 0 \\ 0 & inv(R) \end{pmatrix}$$

Input Parameters

jobu
CHARACTER*1. = 'U': Orthogonal/unitary matrix *U* is computed;
= 'N': *U* is not computed.

jobv
CHARACTER*1. = 'V': Orthogonal/unitary matrix *V* is computed;
= 'N': *V* is not computed.

jobq
CHARACTER*1. = 'Q': Orthogonal/unitary matrix *Q* is computed;
= 'N': *Q* is not computed.

m INTEGER. The number of rows of the matrix *A*.
m≥ 0.

n INTEGER. The number of columns of the matrices *A* and *B*.
n≥ 0.

p INTEGER. The number of rows of the matrix *B*.
p≥ 0.

a REAL for sggsvd3
DOUBLE PRECISION for dggsvd3
COMPLEX for cggsvd3
DOUBLE COMPLEX for zggsvd3
Array, size (*lda*, *n*).
On entry, the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.
lda≥ max(1,*m*).

b REAL for sggsvd3
DOUBLE PRECISION for dggsvd3
COMPLEX for cggsvd3
DOUBLE COMPLEX for zggsvd3
Array, size (*ldb*, *n*).
On entry, the *p*-by-*n* matrix *B*.

ldb INTEGER. The leading dimension of the array *b*.
ldb≥ max(1,*p*).

ldu INTEGER. The leading dimension of the array *u*.
ldu≥ max(1,*m*) if *jobu* = 'U'; *ldu*≥ 1 otherwise.

ldv INTEGER. The leading dimension of the array *v*.
ldv≥ max(1,*p*) if *jobv* = 'V'; *ldv*≥ 1 otherwise.

ldq INTEGER. The leading dimension of the array *q*.
ldq≥ max(1,*n*) if *jobq* = 'Q'; *ldq*≥ 1 otherwise.

work REAL for sggsvd3
DOUBLE PRECISION for dggsvd3
COMPLEX for cggsvd3
DOUBLE COMPLEX for zggsvd3
Array, size (max(1,*lwork*)).

lwork INTEGER. The dimension of the array work.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork

for sggsvd3

for dggsvd3

REAL for cggsvd3

DOUBLE PRECISION for zggsvd3

Array, size (2*n).

iwork

INTEGER. Array, size (n).

Output Parameters

k, *l*

INTEGER. On exit, *k* and *l* specify the dimension of the subblocks described in the Description section.

k + *l* = effective numerical rank of $(A^T,B^T)^T$ for real flavors or $(A^H,B^H)^H$ for complex flavors.

a

On exit, *a* contains the triangular matrix *R*, or part of *R*.

If $m-k-l \geq 0$, *R* is stored in *a*(1: *k* + *l*, *n* - *k* - *l* + 1:*n*).

If $m - k - l < 0$, $\begin{pmatrix} R11 & R12 & R13 \\ 0 & R22 & R23 \end{pmatrix}$ is stored in *a*(1:*m*, *n* - *k* - *l* + 1:*n*), and *R33* is stored in *b*(*m* - *k* + 1:*l*, *n* + *m* - *k* - *l* + 1:*n*) on exit.

b

On exit, *b* contains part of the triangular matrix *R* if $m - k - l < 0$.

See Description for details.

alpha

REAL for sggsvd3

DOUBLE PRECISION for dggsvd3

REAL for cggsvd3

DOUBLE PRECISION for zggsvd3

Array, size (n)

beta

REAL for sggsvd3

DOUBLE PRECISION for dggsvd3

REAL for cggsvd3

DOUBLE PRECISION for zggsvd3

Array, size (n)

On exit, *alpha* and *beta* contain the generalized singular value pairs of *a* and *b*;

alpha(1: *k*) = 1,

beta(1: *k*) = 0,

and if *m* - *k* - *l*≥ 0,

alpha(*k* + 1:*k* + *l*) = *C*,

beta(*k* + 1:*k* + *l*) = *S*,

or if *m* - *k* - *l* < 0,

alpha(*k* + 1:*m*) = *C*, *alpha*(*m* + 1:*k* + *l*) = 0

beta(*k* + 1: *m*) =*S*, *beta*(*m* + 1: *k* + *l*) = 1

and

alpha(*k* + *l* + 1: *n*) = 0

beta(*k* + *l* + 1: *n*) = 0

u

REAL for sggsvd3

DOUBLE PRECISION for dggsvd3

COMPLEX for cggsvd3

DOUBLE COMPLEX for zggsvd3

Array, size (*ldu*, *m*).

If *jobu* = 'U', *u* contains the m-by-m orthogonal/unitary matrix *U*.

If *jobu* = 'N', *u* is not referenced.

v

REAL for sggsvd3

DOUBLE PRECISION for dggsvd3

COMPLEX for cggsvd3

DOUBLE COMPLEX for zggsvd3

Array, size (*ldv*, *p*).

If *jobv* = 'V', *v* contains the *p*-by-*p* orthogonal/unitary matrix *V*.

If *jobv* = 'N', *v* is not referenced.

q

REAL for sggsvd3

DOUBLE PRECISION for dggsvd3

COMPLEX for cggsvd3

DOUBLE COMPLEX for zggsvd3

Array, size (*ldq*, *n*).

If *jobq* = 'Q', *q* contains the *n*-by-*n* orthogonal/unitary matrix *Q*.

If *jobq* = 'N', *q* is not referenced.

work

On exit, if *info* = 0, *work*(1) returns the optimal lwork.

iwork

On exit, *iwork* stores the sorting information. More precisely, the following loop uses *iwork* to sort *alpha*:

```
for I = k+1, min(m,k + l)
    swap alpha(I) and alpha(iwork(I))
endfor
```

such that *alpha*(1) ≥*alpha*(2) ≥ ... ≥*alpha*(*n*).

info INTEGER. = 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

> 0: if *info* = 1, the Jacobi-type procedure failed to converge.

For further details, see subroutine ?tgsja.

Application Notes

?ggsvd3 replaces the deprecated subroutine ?ggsvd.

?tgsja
Computes the generalized SVD of two upper triangular or trapezoidal matrices.

Syntax

call stgsja(*jobu*, *jobv*, *jobq*, *m*, *p*, *n*, *k*, *l*, *a*, *lda*, *b*, *ldb*, *tola*, *tolb*, *alpha*, *beta*, *u*, *ldu*, *v*, *ldv*, *q*, *ldq*, *work*, *ncycle*, *info*)

call dtgsja(*jobu*, *jobv*, *jobq*, *m*, *p*, *n*, *k*, *l*, *a*, *lda*, *b*, *ldb*, *tola*, *tolb*, *alpha*, *beta*, *u*, *ldu*, *v*, *ldv*, *q*, *ldq*, *work*, *ncycle*, *info*)

call ctgsja(*jobu*, *jobv*, *jobq*, *m*, *p*, *n*, *k*, *l*, *a*, *lda*, *b*, *ldb*, *tola*, *tolb*, *alpha*, *beta*, *u*, *ldu*, *v*, *ldv*, *q*, *ldq*, *work*, *ncycle*, *info*)

call ztgsja(*jobu*, *jobv*, *jobq*, *m*, *p*, *n*, *k*, *l*, *a*, *lda*, *b*, *ldb*, *tola*, *tolb*, *alpha*, *beta*, *u*, *ldu*, *v*, *ldv*, *q*, *ldq*, *work*, *ncycle*, *info*)

call tgsja(*a*, *b*, *tola*, *tolb*, *k*, *l* [,*u*] [,*v*] [,*q*] [,*jobu*] [,*jobv*] [,*jobq*] [,*alpha*] [,*beta*] [,*ncycle*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the generalized singular value decomposition (GSVD) of two real/complex upper triangular (or trapezoidal) matrices *A* and *B*. On entry, it is assumed that matrices *A* and *B* have the following forms, which may be obtained by the preprocessing subroutine ggsvp from a general *m*-by-*n* matrix *A* and *p*-by-*n* matrix *B*:

$$A = \begin{array}{r} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} \end{array}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix} \end{array}, \quad \text{if } m-k-l < 0$$

$$B = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

where the *k*-by-*k* matrix A_{12} and *l*-by-*l* matrix B_{13} are nonsingular upper triangular; A_{23} is *l*-by-*l* upper triangular if $m-k-l \geq 0$, otherwise A_{23} is (*m*-*k*)-by-*l* upper trapezoidal.

On exit,

$U^H*A*Q = D_1*(0\ R)$, $V^H*B*Q = D_2*(0\ R)$,

where *U*, *V* and *Q* are orthogonal/unitary matrices, *R* is a nonsingular upper triangular matrix, and D_1 and D_2 are "diagonal" matrices, which are of the following structures:

If $m-k-l \geq 0$,

$$D_1 = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{array}$$

$$D_i = \begin{array}{c} \\ p-l \\ l \end{array} \begin{array}{c} \begin{array}{cc} l & k \end{array} \\ \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{array}$$

$$(0\ R) = \begin{array}{c} \\ l \\ k \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{array}$$

where

$C = \mathrm{diag}(alpha(k+1),...,alpha(k+l))$

$S = \mathrm{diag}(beta(k+1),...,beta(k+l))$

$C^2 + S^2 = I$

R is stored in $a(1{:}k+l,\ n-k-l+1{:}n)$ on exit.

If $m-k-l < 0$,

$$\begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+1-m \end{array} \\ \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ m-k \\ k+l-m \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$(0\ R) = \begin{array}{c} \\ k \\ m-k \\ k+l-m \end{array} \begin{array}{c} \begin{array}{cccc} n-k-l & k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{array}$$

where

C = diag($alpha(k+1),...,alpha(m)$),

S = diag($beta(k+1),...,beta(m)$),

$C^2 + S^2 = I$

On exit,

$$\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$$

is stored in $a(1{:}m, n{-}k{-}l{+}1{:}n)$ and R_{33} is stored

in $b(m{-}k{+}1{:}l, n{+}m{-}k{-}l{+}1{:}n)$.

The computation of the orthogonal/unitary transformation matrices U, V or Q is optional. These matrices may either be formed explicitly, or they *may* be postmultiplied into input matrices U_1, V_1, or Q_1.

Input Parameters

jobu

CHARACTER*1. Must be 'U', 'I', or 'N'.

If *jobu* = 'U', *u* must contain an orthogonal/unitary matrix U_1 on entry.

If *jobu* = 'I', *u* is initialized to the unit matrix.

If *jobu* = 'N', *u* is not computed.

jobv

CHARACTER*1. Must be 'V', 'I', or 'N'.

If *jobv* = 'V', *v* must contain an orthogonal/unitary matrix V_1 on entry.

If *jobv* = 'I', *v* is initialized to the unit matrix.

If *jobv* = 'N', *v* is not computed.

jobq

CHARACTER*1. Must be 'Q', 'I', or 'N'.

If *jobq* = 'Q', *q* must contain an orthogonal/unitary matrix Q_1 on entry.

If *jobq* = 'I', *q* is initialized to the unit matrix.

If *jobq* = 'N', *q* is not computed.

m — INTEGER. The number of rows of the matrix *A* (*m*≥ 0).

p — INTEGER. The number of rows of the matrix *B* (*p*≥ 0).

n — INTEGER. The number of columns of the matrices *A* and *B* (*n*≥ 0).

k, *l* — INTEGER. Specify the subblocks in the input matrices *A* and *B*, whose GSVD is computed.

a, *b*, *u*, *v*, *q*, *work* — REAL for stgsja

DOUBLE PRECISION for dtgsja

COMPLEX for ctgsja

DOUBLE COMPLEX for ztgsja.

Arrays:

a(*lda*,*) contains the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the *p*-by-*n* matrix *B*.

The second dimension of *b* must be at least max(1, *n*).

If *jobu* = 'U', *u*(*ldu*,*) must contain a matrix U_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).

The second dimension of *u* must be at least max(1, *m*).

If *jobv* = 'V', *v*(*ldv*,*) must contain a matrix V_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).

The second dimension of *v* must be at least max(1, *p*).

If *jobq* = 'Q', *q*(*ldq*,*) must contain a matrix Q_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).

The second dimension of *q* must be at least max(1, *n*).

work(*) is a workspace array.

The dimension of *work* must be at least max(1, 2*n*).

lda — INTEGER. The leading dimension of *a*; at least max(1, *m*).

ldb — INTEGER. The leading dimension of *b*; at least max(1, *p*).

ldu — INTEGER. The leading dimension of the array *u* .

ldu≥ max(1, *m*) if *jobu* = 'U'; *ldu*≥ 1 otherwise.

ldv — INTEGER. The leading dimension of the array *v* .

ldv≥ max(1, *p*) if *jobv* = 'V'; *ldv*≥ 1 otherwise.

ldq — INTEGER. The leading dimension of the array *q* .

ldq≥ max(1, *n*) if *jobq* = 'Q'; *ldq*≥ 1 otherwise.

tola, *tolb*

`REAL` for single-precision flavors

`DOUBLE PRECISION` for double-precision flavors.

tola and *tolb* are the convergence criteria for the Jacobi-Kogbetliantz iteration procedure. Generally, they are the same as used in `?ggsvp`:

tola = max(*m*, *n*)*|*A*|*MACHEPS,

tolb = max(*p*, *n*)*|*B*|*MACHEPS.

Output Parameters

a

On exit, *a*(*n*-*k*+1:*n*, 1:min(*k*+*l*, *m*)) contains the triangular matrix *R* or part of *R*.

b

On exit, if necessary, *b*(*m*-*k*+1: *l*, *n*+*m*-*k*-*l*+1: *n*)) contains a part of *R*.

alpha, *beta*

`REAL` for single-precision flavors

`DOUBLE PRECISION` for double-precision flavors.

Arrays, size at least max(1, *n*). Contain the generalized singular value pairs of *A* and *B*:

alpha(1:*k*) = 1,

beta(1:*k*) = 0,

and if *m*-*k*-*l* ≥ 0,

alpha(*k*+1:*k*+*l*) = diag(*C*),

beta(*k*+1:*k*+*l*) = diag(*S*),

or if *m*-*k*-*l* < 0,

alpha(*k*+1:*m*)= diag(*C*), *alpha*(*m*+1:*k*+*l*)=0

beta(*k*+1:*m*) = diag(*S*),

beta(*m*+1:*k*+*l*) = 1.

Furthermore, if *k*+*l* < *n*,

alpha(*k*+*l*+1:*n*)= 0 and

beta(*k*+*l*+1:*n*) = 0.

u

If *jobu* = 'I', *u* contains the orthogonal/unitary matrix *U*.

If *jobu* = 'U', *u* contains the product U_1*U.

If *jobu* = 'N', *u* is not referenced.

v

If *jobv* = 'I', *v* contains the orthogonal/unitary matrix *U*.

If *jobv* = 'V', *v* contains the product V_1*V.

If *jobv* = 'N', *v* is not referenced.

q

If *jobq* = 'I', *q* contains the orthogonal/unitary matrix *U*.

If *jobq* = 'Q', *q* contains the product Q_1*Q.

If *jobq* = 'N', *q* is not referenced.

ncycle INTEGER. The number of cycles required for convergence.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, the procedure does not converge after MAXIT cycles.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine tgsja interface are the following:

a Holds the matrix *A* of size (*m*,*n*).

b Holds the matrix *B* of size (*p*,*n*).

u Holds the matrix *U* of size (*m*,*m*).

v Holds the matrix *V* of size (*p*,*p*).

q Holds the matrix *Q* of size (*n*,*n*).

alpha Holds the vector of length *n*.

beta Holds the vector of length *n*.

jobu If omitted, this argument is restored based on the presence of argument *u* as follows:

jobu = 'U', if *u* is present,

jobu = 'N', if *u* is omitted.

If present, *jobu* must be equal to 'I' or 'U' and the argument *u* must also be present.

Note that there will be an error condition if *jobu* is present and *u* omitted.

jobv If omitted, this argument is restored based on the presence of argument *v* as follows:

jobv = 'V', if *v* is present,

jobv = 'N', if *v* is omitted.

If present, *jobv* must be equal to 'I' or 'V' and the argument *v* must also be present.

Note that there will be an error condition if *jobv* is present and *v* omitted.

jobq If omitted, this argument is restored based on the presence of argument *q* as follows:

jobq = 'Q', if *q* is present,

jobq = 'N', if *q* is omitted.

If present, *jobq* must be equal to 'I' or 'Q' and the argument *q* must also be present.

Note that there will be an error condition if *jobq* is present and *q* omitted.

Cosine-Sine Decomposition: LAPACK Computational Routines

This topic describes LAPACK computational routines for computing the *cosine-sine decomposition* (CS decomposition) of a partitioned unitary/orthogonal matrix. The algorithm computes a complete 2-by-2 CS decomposition, which requires simultaneous diagonalization of all the four blocks of a unitary/orthogonal matrix partitioned into a 2-by-2 block structure.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

Table "Computational Routines for Cosine-Sine Decomposition (CSD)" lists LAPACK routines that perform CS decomposition of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Cosine-Sine Decomposition (CSD)

Operation	Real matrices	Complex matrices
Compute the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form	bbcsd/bbcsd	bbcsd/bbcsd
Simultaneously bidiagonalize the blocks of a partitioned orthogonal matrix	orbdb unbdb	
Simultaneously bidiagonalize the blocks of a partitioned unitary matrix		orbdb unbdb

See Also

CS Driver Routine

?bbcsd
Computes the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form.

Syntax

```
call sbbcsd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, work,
lwork, info )

call dbbcsd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, work,
lwork, info )

call cbbcsd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, rwork,
rlwork, info )

call zbbcsd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, rwork,
rlwork, info )

call bbcsd( theta,phi,u1,u2,v1t,v2t[,b11d][,b11e][,b12d][,b12e][,b21d][,b21e][,b22d]
[,b22e][,jobu1][,jobu2][,jobv1t][,jobv2t][,trans][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

mkl_lapack.fiThe routine ?bbcsd computes the CS decomposition of an orthogonal or unitary matrix in bidiagonal-block form:

$$X=\begin{pmatrix} b_{11} & b_{12} & 0 & 0 \\ 0 & 0 & -I & 0 \\ \hline b_{21} & b_{22} & 0 & 0 \\ 0 & 0 & 0 & I \end{pmatrix}=\begin{pmatrix} u_1 & \\ & u_2 \end{pmatrix}\begin{pmatrix} C & -S & 0 & 0 \\ 0 & 0 & -I & 0 \\ \hline S & C & 0 & 0 \\ 0 & 0 & 0 & I \end{pmatrix}\begin{pmatrix} v_1 & \\ & v_2 \end{pmatrix}^T$$

or

$$X=\begin{pmatrix} b_{11} & b_{12} & 0 & 0 \\ 0 & 0 & -I & 0 \\ \hline b_{21} & b_{22} & 0 & 0 \\ 0 & 0 & 0 & I \end{pmatrix}=\begin{pmatrix} u_1 & \\ & u_2 \end{pmatrix}\begin{pmatrix} C & -S & 0 & 0 \\ 0 & 0 & -I & 0 \\ \hline S & C & 0 & 0 \\ 0 & 0 & 0 & I \end{pmatrix}\begin{pmatrix} v_1 & \\ & v_2 \end{pmatrix}^H$$

respectively.

x is *m*-by-*m* with the top-left block *p*-by-*q*. Note that *q* must not be larger than *p*, *m*-*p*, or *m*-*q*. If *q* is not the smallest index, *x* must be transposed and/or permuted in constant time using the *trans* option. See ?orcsd/?uncsd for details.

The bidiagonal matrices b_{11}, b_{12}, b_{21}, and b_{22} are represented implicitly by angles *theta*(1:*q*) and *phi*(1:*q*-1).

The orthogonal/unitary matrices u_1, u_2, v_1^t, and v_2^t are input/output. The input matrices are pre- or post-multiplied by the appropriate singular vector matrices.

Input Parameters

jobu1 — CHARACTER. If equals *Y*, then u_1 is updated. Otherwise, u_1 is not updated.

jobu2 — CHARACTER. If equals *Y*, then u_2 is updated. Otherwise, u_2 is not updated.

jobv1t — CHARACTER. If equals *Y*, then v_1^t is updated. Otherwise, v_1^t is not updated.

jobv2t — CHARACTER. If equals *Y*, then v_2^t is updated. Otherwise, v_2^t is not updated.

trans — CHARACTER

= 'T':	*x*, u_1, u_2, v_1^t, v_2^t are stored in row-major order.
otherwise	*x*, u_1, u_2, v_1^t, v_2^t are stored in column-major order.

m — INTEGER. The number of rows and columns of the orthogonal/unitary matrix *X* in bidiagonal-block form.

p — INTEGER. The number of rows in the top-left block of *x*. 0 ≤*p*≤*m*.

≤

q INTEGER. The number of columns in the top-left block of *x*. $0 \le q \le \min(p, m-p, m-q)$.

theta REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*q*).

On entry, the angles *theta*(1), ..., *theta*(*q*) that, along with *phi*(1), ..., *phi*(*q*-1), define the matrix in bidiagonal-block form as returned by orbdb/unbdb.

phi REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*q*-1).

The angles *phi*(1), ..., *phi*(*q*-1) that, along with *theta*(1), ..., *theta*(*q*), define the matrix in bidiagonal-block form as returned by orbdb/unbdb.

u1 REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*ldu1*,*p*).

On entry, a *p*-by-*p* matrix.

ldu1 INTEGER. The leading dimension of the array u_1, $ldu1 \le \max(1, p)$.

u2 REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*ldu2*,*m*-*p*).

On entry, an (*m*-*p*)-by-(*m*-*p*) matrix.

ldu2 INTEGER. The leading dimension of the array u_2, $ldu2 \le \max(1, m-p)$.

v1t REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*ldv1t*,*q*).

On entry, a *q*-by-*q* matrix.

ldv1t INTEGER. The leading dimension of the array *v1t*, *ldv1t*≤ max(1, *q*).

v2t REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*ldv2t*,*m*-*q*).

On entry, an (*m*-*q*)-by-(*m*-*q*) matrix.

ldv2t INTEGER. The leading dimension of the array *v2t*, *ldv2t*≤ max(1, *m*-*q*).

work REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Workspace array, size (max(1,*lwork*)).

lwork INTEGER. The size of the *work* array. *lwork*≥ max(1,8*q)

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

theta REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

On exit, the angles whose cosines and sines define the diagonal blocks in the CS decomposition.

u1 REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

On exit, *u1* is postmultiplied by the left singular vector matrix common to [b11 ; 0] and [b12 0 0 ; 0 -I 0].

u2 REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

On exit, *u2* is postmultiplied by the left singular vector matrix common to [b21 ; 0] and [b22 0 0 ; 0 0 I].

v1t

REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*q*).

On exit, *v1t* is premultiplied by the transpose of the right singular vector matrix common to [b11 ; 0] and [b21 ; 0].

v2t

REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

On exit, *v2t* is premultiplied by the transpose of the right singular vector matrix common to [b12 0 0 ; 0 -I 0] and [b22 0 0 ; 0 0 I].

b11d

REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*q*).

When ?bbcsd converges, *b11d* contains the cosines of *theta*(1), ..., *theta*(*q*). If ?bbcsd fails to converge, *b11d* contains the diagonal of the partially reduced top left block.

b11e

REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*q*-1).

When ?bbcsd converges, *b11e* contains zeros. If ?bbcsd fails to converge, *b11e* contains the superdiagonal of the partially reduced top left block.

b12d

REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*q*).

When ?bbcsd converges, *b12d* contains the negative sines of *theta*(1), ..., *theta*(*q*). If ?bbcsd fails to converge, *b12d* contains the diagonal of the partially reduced top right block.

b12e

REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, size (*q*-1).

When ?bbcsd converges, *b12e* contains zeros. If ?bbcsd fails to converge, *b11e* contains the superdiagonal of the partially reduced top right block.

info

INTEGER.

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument has an illegal value

> 0: if ?bbcsd did not converge, *info* specifies the number of nonzero entries in *phi*, and *b11d*, *b11e*, etc. contain the partially reduced matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran 95 Interface Conventions.

Specific details for the routine ?bbcsd interface are as follows:

theta Holds the vector of length *q*.

phi Holds the vector of length *q*-1.

u1 Holds the matrix of size (*p*,*p*).

u2 Holds the matrix of size (*m*-*p*,*m*-*p*).

v1t Holds the matrix of size (*q*,*q*).

v2t Holds the matrix of size (*m*-*q*,*m*-*q*).

b11d Holds the vector of length *q*.

b11e Holds the vector of length *q*-1.

b12d Holds the vector of length *q*.

b12e Holds the vector of length *q*-1.

b21d Holds the vector of length *q*.

b21e Holds the vector of length *q*-1.

b22d Holds the vector of length *q*.

b22e Holds the vector of length *q*-1.

jobsu1 Indicates whether u_1 is computed. Must be 'Y' or 'O'.

jobsu2	Indicates whether u_2 is computed. Must be `'Y'` or `'O'`.
jobv1t	Indicates whether v_1^t is computed. Must be `'Y'` or `'O'`.
jobv2t	Indicates whether v_2^t is computed. Must be `'Y'` or `'O'`.
trans	Must be `'N'` or `'T'`.

See Also

?orcsd/?uncsd

xerbla

?orbdb/?unbdb

Simultaneously bidiagonalizes the blocks of a partitioned orthogonal/unitary matrix.

Syntax

```
call sorbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
```

```
call dorbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
```

```
call cunbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
```

```
call zunbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
```

```
call orbdb( x11,x12,x21,x22,theta,phi,taup1,taup2,tauq1,tauq2[,trans][,signs][,info] )
```

```
call unbdb( x11,x12,x21,x22,theta,phi,taup1,taup2,tauq1,tauq2[,trans][,signs][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines ?orbdb/?unbdb simultaneously bidiagonalizes the blocks of an *m*-by-*m* partitioned orthogonal matrix *X*:

$$X=\left(\begin{array}{c|c} x_{11} & x_{12} \\ \hline x_{21} & x_{22} \end{array}\right)=\left(\begin{array}{c|c} p_1 & \\ \hline & p_2 \end{array}\right)\left(\begin{array}{c|c|c|c} b_{11} & b_{12} & 0 & 0 \\ \hline 0 & 0 & -I & 0 \\ \hline b_{21} & b_{22} & 0 & 0 \\ \hline 0 & 0 & 0 & I \end{array}\right)\left(\begin{array}{c|c} q_1 & \\ \hline & q_2 \end{array}\right)^T$$

or unitary matrix:

$$X=\left(\begin{array}{c|c} x_{11} & x_{12} \\ \hline x_{21} & x_{22} \end{array}\right)=\left(\begin{array}{c|c} p_1 & \\ \hline & p_2 \end{array}\right)\left(\begin{array}{c|c|c|c} b_{11} & b_{12} & 0 & 0 \\ \hline 0 & 0 & -I & 0 \\ \hline b_{21} & b_{22} & 0 & 0 \\ \hline 0 & 0 & 0 & I \end{array}\right)\left(\begin{array}{c|c} q_1 & \\ \hline & q_2 \end{array}\right)^H$$

x_{11} is p-by-q. q must not be larger than p, m-p, or m-q. Otherwise, x must be transposed and/or permuted in constant time using the *trans* and *signs* options.

The orthogonal/unitary matrices p_1, p_2, q_1, and q_2 are p-by-p, $(m-p)$-by-$(m-p)$, q-by-q, $(m-q)$-by-$(m-q)$, respectively. They are represented implicitly by Housholder vectors.

The bidiagonal matrices b_{11}, b_{12}, b_{21}, and b_{22} are q-by-q bidiagonal matrices represented implicitly by angles *theta*(1), ..., *theta*(q) and *phi*(1), ..., *phi*(q-1). b_{11} and b_{12} are upper bidiagonal, while b_{21} and b_{22} are lower bidiagonal. Every entry in each bidiagonal band is a product of a sine or cosine of *theta* with a sine or cosine of *phi*. See [Sutton09] for details.

p_1, p_2, q_1, and q_2 are represented as products of elementary reflectors. .

Input Parameters

trans

CHARACTER

= 'T':	x, u_1, u_2, v_1^t, v_2^t are stored in row-major order.
otherwise	x, u_1, u_2, v_1^t, v_2^t are stored in column-major order.

signs

CHARACTER

= 'O':	The lower-left block is made nonpositive (the "other" convention).
otherwise	The upper-right block is made nonpositive (the "default" convention).

m

INTEGER. The number of rows and columns of the matrix X.

p

INTEGER. The number of rows in x_{11} and x_{12}. $0 \le p \le m$.

q

INTEGER. The number of columns in x_{11} and x_{21}. $0 \le q \le \min(p, m-p, m-q)$.

x11

REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*ldx11*,*) .

On entry, the top-left block of the orthogonal/unitary matrix to be reduced.

ldx11

INTEGER. The leading dimension of the array X_{11}. If *trans* = 'T', $ldx11 \ge p$. Otherwise, $ldx11 \ge q$.

x12

REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*ldx12*,*m*-*q*).

On entry, the top-right block of the orthogonal/unitary matrix to be reduced.

ldx12 INTEGER. The leading dimension of the array X_{12}. If *trans* = 'N', *ldx12*≥*p*. Otherwise, *ldx12*≥*m*-*q*.

x21 REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*ldx21*,*q*).

On entry, the bottom-left block of the orthogonal/unitary matrix to be reduced.

ldx21 INTEGER. The leading dimension of the array X_{21}. If *trans* = 'N', *ldx21*≥*m*-*p*. Otherwise, *ldx21*≥*q*.

x22 REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*ldx22*,*m*-*q*).

On entry, the bottom-right block of the orthogonal/unitary matrix to be reduced.

ldx22 INTEGER. The leading dimension of the array X_{21}. If *trans* = 'N', *ldx22*≥*m*-*p*. Otherwise, *ldx22*≥*m*-*q*.

work REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Workspace array, size (*lwork*).

lwork INTEGER. The size of the *work* array. *lwork*≥*m*-*q*

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x11 On exit, the form depends on *trans*:

If *trans*='N',	the columns of the lower triangle of *x11* specify reflectors for p_1, the rows of the upper triangle of *x11*(1:*q* - 1, *q*:q - 1) specify reflectors for q_1
otherwise *trans*='T',	the rows of the upper triangle of *x11* specify reflectors for p_1, the columns of the lower triangle of *x11*(1:*q* - 1, *q*:q - 1) specify reflectors for q_1

x12

On exit, the form depends on *trans*:

If *trans*='T',	the columns of the upper triangle of *x12* specify the first *p* reflectors for q_2
otherwise *trans*='T',	the columns of the lower triangle of *x12* specify the first *p* reflectors for q_2

x21

On exit, the form depends on *trans*:

If *trans*='T',	the columns of the lower triangle of *x21* specify the reflectors for p_2
otherwise *trans*='T',	the columns of the upper triangle of *x21* specify the reflectors for p_2

x22

On exit, the form depends on *trans*:

If *trans*='T',	the rows of the upper triangle of *x22*(*q*+1:*m*-*p*,*p*+1:*m*-*q*) specify the last *m*-*p*-*q* reflectors for q_2
otherwise *trans*='T',	the columns of the lower triangle of *x22*(*p*+1:*m*-*q*,*q*+1:*m*-*p*) specify the last *m*-*p*-*q* reflectors for p_2

theta

REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*q*). The entries of bidiagonal blocks b_{11}, b_{12}, b_{21}, and b_{22} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi

REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*q*-1). The entries of bidiagonal blocks b_{11}, b_{12}, b_{21}, and b_{22} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1

REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*p*).

Scalar factors of the elementary reflectors that define p_1.

taup2

REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*m*-*p*).

Scalar factors of the elementary reflectors that define p_2.

tauq1

REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*q*).

Scalar factors of the elementary reflectors that define q_1.

tauq2

REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (*m*-*q*).

Scalar factors of the elementary reflectors that define q_2.

info

INTEGER.

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument has an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran 95 Interface Conventions.

Specific details for the routine ?orbdb/?unbdb interface are as follows:

x11 Holds the block of matrix *X* of size (*p*, *q*).

x12 Holds the block of matrix *X* of size (*p*, *m*-*q*).

x21 Holds the block of matrix *X* of size (*m*-*p*, *q*).

x22 Holds the block of matrix *X* of size (*m*-*p*, *m*-*q*).

theta Holds the vector of length *q*.

phi Holds the vector of length *q*-1.

taup1 Holds the vector of length *p*.

taup2 Holds the vector of length *m*-*p*.

tauq1 Holds the vector of length *q*.

taupq2 Holds the vector of length *m*-*q*.

trans Must be 'N' or 'T'.

signs Must be 'O' or 'D'.

See Also

?orcsd/?uncsd

?orgqr

?ungqr

?orglq

?unglq

xerbla

LAPACK Least Squares and Eigenvalue Problem Driver Routines

Each of the LAPACK driver routines solves a complete problem. To arrive at the solution, driver routines typically call a sequence of appropriate computational routines.

Driver routines are described in the following topics :

Linear Least Squares (LLS) Problems

Generalized LLS Problems

Symmetric Eigenproblems

Nonsymmetric Eigenproblems

Singular Value Decomposition

Cosine-Sine Decomposition

Generalized Symmetric Definite Eigenproblems

Generalized Nonsymmetric Eigenproblems

Linear Least Squares (LLS) Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving linear least squares problems. Table "Driver Routines for Solving LLS Problems" lists all such routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving LLS Problems

Routine Name	Operation performed
gels	Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.
gelsy	Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.
gelss	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.
gelsd	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.
getsls	Solves overdetermined or underdetermined real linear systems involving a matrix or its transpose using a tall skinny QR or short wide LQ factorization.

?gels
Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.

Syntax

```
call sgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
```

```
call dgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call cgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call zgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call gels(a, b [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves overdetermined or underdetermined real/ complex linear systems involving an *m*-by-*n* matrix *A*, or its transpose/ conjugate-transpose, using a *QR* or *LQ* factorization of *A*. It is assumed that *A* has full rank.

The following options are provided:

1. If *trans* = 'N' and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

minimize $||b - A*x||_2$

2. If *trans* = 'N' and $m < n$: find the minimum norm solution of an underdetermined system $A*X = B$.

3. If *trans* = 'T' or 'C' and $m \geq n$: find the minimum norm solution of an undetermined system $A^H*X = B$.

4. If *trans* = 'T' or 'C' and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

minimize $||b - A^H*x||_2$

Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; they are formed by the columns of the right hand side matrix *B* and the solution matrix *X* (when coefficient matrix is *A*, *B* is *m*-by-*nrhs* and *X* is *n*-by-*nrhs*; if the coefficient matrix is A^T or A^H, *B* is*n*-by-*nrhs* and *X* is *m*-by-*nrhs*.

Input Parameters

trans	CHARACTER*1. Must be 'N', 'T', or 'C'. If *trans* = 'N', the linear system involves matrix *A*; If *trans* = 'T', the linear system involves the transposed matrix A^T (for real flavors only); If *trans* = 'C', the linear system involves the conjugate-transposed matrix A^H (for complex flavors only).
m	INTEGER. The number of rows of the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns of the matrix *A* ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* ($nrhs \geq 0$).
a, *b*, *work*	REAL for sgels DOUBLE PRECISION for dgels COMPLEX for cgels

DOUBLE COMPLEX for zgels.

Arrays:

a(*lda*,*) contains the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the matrix *B* of right hand side vectors.

The second dimension of *b* must be at least max(1, *nrhs*).

work is a workspace array, its dimension max(1, *lwork*).

lda INTEGER. The leading dimension of *a*; at least max(1, *m*).

ldb INTEGER. The leading dimension of *b*; must be at least max(1, *m*, *n*).

lwork INTEGER. The size of the *work* array; must be at least min (*m*, *n*)+max(1, *m*, *n*, *nrhs*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a On exit, overwritten by the factorization data as follows:

if *m*≥*n*, array *a* contains the details of the *QR* factorization of the matrix *A* as returned by ?geqrf;

if *m* < *n*, array *a* contains the details of the *LQ* factorization of the matrix *A* as returned by ?gelqf.

b If *info* = 0, *b* overwritten by the solution vectors, stored columnwise:

if *trans* = 'N' and *m*≥*n*, rows 1 to *n* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements *n*+1 to *m* in that column;

if *trans* = 'N' and *m* < *n*, rows 1 to *n* of *b* contain the minimum norm solution vectors;

if *trans* = 'T' or 'C' and *m*≥*n*, rows 1 to *m* of *b* contain the minimum norm solution vectors;

if *trans* = 'T' or 'C' and *m* < *n*, rows 1 to *m* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements *m*+1 to *n* in that column.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of the triangular factor of *A* is zero, so that *A* does not have full rank; the least squares solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gels` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
b	Holds the matrix of size max(*m*,*n*)-by-*nrhs*. If *trans* = 'N', then, on entry, the size of *b* is *m*-by-*nrhs*, If *trans* = 'T', then, on entry, the size of *b* is *n*-by-*nrhs*,
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = min (*m*, *n*)+max(1, *m*, *n*, *nrhs*)***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelsy

Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.

Syntax

```
call sgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, info)
call dgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, info)
call cgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, rwork, info)
call zgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, rwork, info)
call gelsy(a, b [,rank] [,jpvt] [,rcond] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The `?gelsy` routine computes the minimum-norm solution to a real/complex linear least squares problem:

`minimize ||b - A*x||`$_2$

using a complete orthogonal factorization of *A*. *A* is an *m*-by-*n* matrix which may be rank-deficient. Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; they are stored as the columns of the *m*-by-*nrhs* right hand side matrix *B* and the *n*-by-*nrhs* solution matrix *X*.

The routine first computes a *QR* factorization with column pivoting:

$$AP = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

with R_{11} defined as the largest leading submatrix whose estimated condition number is less than 1/*rcond*. The order of R_{11}, *rank*, is the effective rank of *A*. Then, R_{22} is considered to be negligible, and R_{12} is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

The minimum-norm solution is then

$$X = PZ^{T} \begin{pmatrix} T_{11}^{-1} Q_1^{T} B \\ 0 \end{pmatrix}$$

for real flavors and

$$X = PZ^{H} \begin{pmatrix} T_{11}^{-1} Q_1^{H} B \\ 0 \end{pmatrix}$$

for complex flavors,

where Q_1 consists of the first *rank* columns of *Q*.

The ?gelsy routine is identical to the original deprecated ?gelsx routine except for the following differences:

- The call to the subroutine ?geqpf has been substituted by the call to the subroutine ?geqp3, which is a BLAS-3 version of the *QR* factorization with column pivoting.
- The matrix *B* (the right hand side) is updated with BLAS-3.
- The permutation of the matrix *B* (the right hand side) is faster and more simple.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns of the matrix *A* (*n*≥ 0).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs*≥ 0).
a, *b*, *work*	REAL for sgelsy DOUBLE PRECISION for dgelsy COMPLEX for cgelsy DOUBLE COMPLEX for zgelsy. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *m*-by-*nrhs* right hand side matrix *B*. The second dimension of *b* must be at least max(1, *nrhs*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The leading dimension of *b*; must be at least max(1, *m*, *n*).
jpvt	INTEGER. Array, size at least max(1, *n*). On entry, if *jpvt*(*i*)≠ 0, the *i*-th column of *A* is permuted to the front of *AP*, otherwise the *i*-th column of *A* is a free column.
rcond	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. *rcond* is used to determine the effective rank of *A*, which is defined as the order of the largest leading triangular submatrix R_{11} in the *QR* factorization with pivoting of *A*, whose estimated condition number < 1/*rcond*.
lwork	INTEGER. The size of the *work* array. If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.

rwork — REAL for cgelsy DOUBLE PRECISION for zgelsy. Workspace array, size at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

a — On exit, overwritten by the details of the complete orthogonal factorization of *A*.

b — Overwritten by the *n*-by-*nrhs* solution matrix *X*.

jpvt — On exit, if *jpvt*(*i*)= *k*, then the *i*-th column of *AP* was the *k*-th column of *A*.

rank — INTEGER. The effective rank of *A*, that is, the order of the submatrix R_{11}. This is the same as the order of the submatrix T_{11} in the complete orthogonal factorization of *A*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gelsy interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

b — Holds the matrix of size max(*m*,*n*)-by-*nrhs*. On entry, contains the *m*-by-*nrhs* right hand side matrix *B*, On exit, overwritten by the *n*-by-*nrhs* solution matrix *X*.

jpvt — Holds the vector of length *n*. Default value for this element is *jpvt*(*i*) = 0.

rcond — Default value for this element is *rcond* = 100*EPSILON(1.0_WP).

Application Notes

For real flavors:

The unblocked strategy requires that:

lwork≥ max(*mn*+3*n*+1, 2**mn* + *nrhs*),

where *mn* = min(*m*, *n*).

The block algorithm requires that:

lwork≥ max(*mn*+2*n*+*nb**(*n*+1), 2**mn*+*nb***nrhs*),

where *nb* is an upper bound on the blocksize returned by ilaenv for the routines sgeqp3/dgeqp3, stzrzf/dtzrzf, stzrqf/dtzrqf, sormqr/dormqr, and sormrz/dormrz.

For complex flavors:

The unblocked strategy requires that:

lwork≥*mn* + max(2**mn*, *n*+1, *mn* + *nrhs*),

where *mn* = min(*m*, *n*).

The block algorithm requires that:

lwork < *mn* + max(2**mn*, *nb**(*n*+1), *mn*+*mn***nb*, *mn*+ *nb***nrhs*),

where *nb* is an upper bound on the blocksize returned by ilaenv for the routines cgeqp3/zgeqp3, ctzrzf/ztzrzf, ctzrqf/ztzrqf, cunmqr/zunmqr, and cunmrz/zunmrz.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelss

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.

Syntax

call sgelss(*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*, *work*, *lwork*, *info*)

call dgelss(*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*, *work*, *lwork*, *info*)

call cgelss(*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*, *work*, *lwork*, *rwork*, *info*)

call zgelss(*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*, *work*, *lwork*, *rwork*, *info*)

call gelss(*a*, *b* [,*rank*] [,*s*] [,*rcond*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the minimum norm solution to a real linear least squares problem:

minimize $||b - A*x||_2$

using the singular value decomposition (SVD) of *A*. *A* is an *m*-by-*n* matrix which may be rank-deficient. Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; they are stored as the columns of the *m*-by-*nrhs* right hand side matrix *B* and the *n*-by-*nrhs* solution matrix *X*. The effective rank of *A* is determined by treating as zero those singular values which are less than *rcond* times the largest singular value.

Input Parameters

m — INTEGER. The number of rows of the matrix *A* (*m*≥ 0).

n — INTEGER. The number of columns of the matrix *A* (*n*≥ 0).

nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs*≥ 0).
a, *b*, *work*	REAL for sgelss DOUBLE PRECISION for dgelss COMPLEX for cgelss DOUBLE COMPLEX for zgelss. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *m*-by-*nrhs* right hand side matrix *B*. The second dimension of *b* must be at least max(1, *nrhs*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The leading dimension of *b*; must be at least max(1, *m*, *n*).
rcond	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. *rcond* is used to determine the effective rank of *A*. Singular values *s*(*i*) ≤*rcond* **s*(1) are treated as zero. If *rcond* <0, machine precision is used instead.
lwork	INTEGER. The size of the *work* array; *lwork*≥ 1. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for the suggested value of *lwork*.
rwork	REAL for cgelss DOUBLE PRECISION for zgelss. Workspace array used in complex flavors only. size at least max(1, 5*min(*m*, *n*)).

Output Parameters

a	On exit, the first min(*m*, *n*) rows of *a* are overwritten with the matrix of right singular vectors of *A*, stored row-wise.
b	Overwritten by the *n*-by-*nrhs* solution matrix *X*. If *m*≥*n* and *rank* = *n*, the residual sum-of-squares for the solution in the *i*-th column is given by the sum of squares of modulus of elements *n*+1:*m* in that column.
s	REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, min(*m*, *n*)). The singular values of *A* in decreasing order. The condition number of *A* in the 2-norm is

$k_2(A) = s(1) / s(\min(m, n))$.

rank — INTEGER. The effective rank of *A*, that is, the number of singular values which are greater than *rcond* **s*(1).

work(1) — If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm for computing the SVD failed to converge; *i* indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gelss interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

b — Holds the matrix of size max(*m*,*n*)-by-*nrhs*. On entry, contains the *m*-by-*nrhs* right hand side matrix *B*, On exit, overwritten by the *n*-by-*nrhs* solution matrix *X*.

s — Holds the vector of length min(*m*,*n*).

rcond — Default value for this element is *rcond* = 100*EPSILON(1.0_WP).

Application Notes

For real flavors:

lwork≥ 3*min(*m*, *n*)+ max(2*min(*m*, *n*), max(*m*, *n*), *nrhs*)

For complex flavors:

lwork≥ 2*min(*m*, *n*)+ max(*m*, *n*, *nrhs*)

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelsd

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

Syntax

```
call sgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info)
call dgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info)
call cgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, iwork, info)
call zgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, iwork, info)
call gelsd(a, b [,rank] [,s] [,rcond] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the minimum-norm solution to a real linear least squares problem:

$\text{minimize } ||b - A*x||_2$

using the singular value decomposition (SVD) of *A*. *A* is an *m*-by-*n* matrix which may be rank-deficient.

Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; they are stored as the columns of the *m*-by-*nrhs* right hand side matrix *B* and the *n*-by-*nrhs* solution matrix *X*.

The problem is solved in three steps:

1. Reduce the coefficient matrix A to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).
2. Solve the BLS using a divide and conquer approach.
3. Apply back all the Householder transformations to solve the original least squares problem.

The effective rank of *A* is determined by treating as zero those singular values which are less than *rcond* times the largest singular value.

The routine uses auxiliary routines lals0 and lalsa.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns of the matrix *A* (*n*≥ 0).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs*≥ 0).
a, *b*, *work*	REAL for sgelsd DOUBLE PRECISION for dgelsd COMPLEX for cgelsd

DOUBLE COMPLEX for zgelsd.

Arrays:

a(*lda*,*) contains the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the *m*-by-*nrhs* right hand side matrix *B*.

The second dimension of *b* must be at least max(1, *nrhs*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; at least max(1, *m*).

ldb — INTEGER. The leading dimension of *b*; must be at least max(1, *m*, *n*).

rcond — REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

rcond is used to determine the effective rank of *A*. Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If $rcond \leq 0$, machine precision is used instead.

lwork — INTEGER. The size of the *work* array; $lwork \geq 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the array *work* and the minimum sizes of the arrays *rwork* and *iwork*, and returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

iwork — INTEGER. Workspace array. See *Application Notes* for the suggested dimension of *iwork*.

rwork — REAL for cgelsd

DOUBLE PRECISION for zgelsd.

Workspace array, used in complex flavors only. See *Application Notes* for the suggested dimension of *rwork*.

Output Parameters

a — On exit, *A* has been overwritten.

b — Overwritten by the *n*-by-*nrhs* solution matrix *X*.

If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the *i*-th column is given by the sum of squares of modulus of elements *n*+1:*m* in that column.

s — REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, min(*m*, *n*)). The singular values of *A* in decreasing order. The condition number of *A* in the 2-norm is

$k_2(A) = s(1) / s(\min(m, n))$.

rank	INTEGER. The effective rank of *A*, that is, the number of singular values which are greater than *rcond* **s*(1).
work(1)	If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
rwork(1)	If *info* = 0, on exit, *rwork*(1) returns the minimum size of the workspace array *iwork* required for optimum performance.
iwork(1)	If *info* = 0, on exit, *iwork*(1) returns the minimum size of the workspace array *iwork* required for optimum performance.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then the algorithm for computing the SVD failed to converge; *i* indicates the number of off-diagonal elements of an intermediate bidiagonal form that did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gelsd interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
b	Holds the matrix of size max(*m*,*n*)-by-*nrhs*. On entry, contains the *m*-by-*nrhs* right hand side matrix *B*, On exit, overwritten by the *n*-by-*nrhs* solution matrix *X*.
s	Holds the vector of length min(*m*,*n*).
rcond	Default value for this element is *rcond* = 100*EPSILON(1.0_WP).

Application Notes

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

The exact minimum amount of workspace needed depends on *m*, *n* and *nrhs*. The size *lwork* of the workspace array *work* must be as given below.

For real flavors:

If $m \geq n$,

$lwork \geq 12n + 2n*smlsiz + 8n*nlvl + n*nrhs + (smlsiz+1)^2$;

If $m < n$,

$lwork \geq 12m + 2m*smlsiz + 8m*nlvl + m*nrhs + (smlsiz+1)^2$;

For complex flavors:

If $m \geq n$,

$lwork < 2n + n*nrhs$;

If $m < n$,

$lwork \geq 2m + m*nrhs$;

where *smlsiz* is returned by `ilaenv` and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and

$nlvl = \mathrm{INT}(\log_2(\min(m, n)/(smlsiz+1))) + 1$.

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The dimension of the workspace array *iwork* must be at least

$3*\min(m, n)*nlvl + 11*\min(m, n)$.

The dimension of the workspace array *iwork* (for complex flavors) must be at least $\max(1, lrwork)$.

$lrwork \geq 10n + 2n*smlsiz + 8n*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2$ if $m \geq n$, and

$lrwork \geq 10m + 2m*smlsiz + 8m*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2$ if $m < n$.

?getsls
Uses QR or LQ factorization to solve an overdetermined or underdetermined linear system with full rank matrix, with best performance for tall and skinny matrices.

```
call sgetsls(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call dgetsls(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call cgetsls(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call zgetsls(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
```

Description

The routine solves overdetermined or underdetermined real/ complex linear systems involving an *m*-by-*n* matrix *A*, or its transpose/conjugate-transpose, using a `?geqr` or `?gelq` factorization of *A*. It is assumed that *A* has full rank.

The following options are provided:

1. If $trans = \texttt{'N'}$ and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$\text{minimize } ||b - A*x||_2$

2. If $trans = \texttt{'N'}$ and $m < n$: find the minimum norm solution of an underdetermined system $A*X = B$.

3. If $trans = \texttt{'T'}$ or `'C'` and $m \geq n$: find the minimum norm solution of an undetermined system $A^H*X = B$.

4. If *trans* = 'T' or 'C' and *m* < *n*: find the least squares solution of an overdetermined system, that is, solve the least squares problem

minimize $||b - A^H * x||_2$

Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; they are formed by the columns of the right hand side matrix *B* and the solution matrix *X* (when coefficient matrix is *A*, *B* is *m*-by-*nrhs* and *X* is *n*-by-*nrhs*; if the coefficient matrix is A^T or A^H, *B* is*n*-by-*nrhs* and *X* is *m*-by-*nrhs*.

Input Parameters

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.

If *trans* = 'N', the linear system involves matrix *A*;

If *trans* = 'T', the linear system involves the transposed matrix A^T (for real flavors only);

If *trans* = 'C', the linear system involves the conjugate-transposed matrix A^H (for complex flavors only).

m

INTEGER. The number of rows of the matrix *A*. *m*≥ 0.

n

INTEGER. The number of columns of the matrix *A*. *n*≥ 0.

nrhs

INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs*≥ 0).

a

REAL for sgetsls

DOUBLE PRECISION for dgetsls

COMPLEX for cgetsls

COMPLEX*16 for zgetsls

Array *a*(*lda*,*) contains the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

lda

INTEGER. The leading dimension of the array *a*. *lda*≥max(1,*m*).

b

REAL for sgetsls

DOUBLE PRECISION for dgetsls

COMPLEX for cgetsls

COMPLEX*16 for zgetsls

Array *b*(*ldb*,*) contains the matrix *B* of right hand side vectors.

The second dimension of *b* must be at least max(1, *nrhs*).

ldb

INTEGER. The leading dimension of the array *b*. *ldb*≥max(1,*m*,*n*).

lwork

INTEGER. The size of the *work* array; must be at least min (*m*, *n*)+max(1, *m*, *n*, *nrhs*).

If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a	On exit, overwritten by the factorization data as follows: if *m*≥*n*, array *a* contains the details of the *QR* factorization of the matrix *A* as returned by ?geqr; if *m* < *n*, array *a* contains the details of the *LQ* factorization of the matrix *A* as returned by ?gelq.
b	If *info* = 0, *b* overwritten by the solution vectors, stored columnwise: if *trans* = 'N' and *m*≥*n*, rows 1 to *n* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements *n*+1 to *m* in that column; if *trans* = 'N' and *m* < *n*, rows 1 to *n* of *b* contain the minimum norm solution vectors; if *trans* = 'T' or 'C' and *m*≥*n*, rows 1 to *m* of *b* contain the minimum norm solution vectors; if *trans* = 'T' or 'C' and *m* < *n*, rows 1 to *m* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements *m*+1 to *n* in that column.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of the triangular factor of *A* is zero, so that *A* does not have full rank; the least squares solution could not be computed.

Generalized Linear Least Squares (LLS) Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving generalized linear least squares problems. Table "Driver Routines for Solving Generalized LLS Problems" lists all such routines. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Generalized LLS Problems

Routine Name	Operation performed
gglse	Solves the linear equality-constrained least squares problem using a generalized RQ factorization.
ggglm	Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

?gglse
Solves the linear equality-constrained least squares problem using a generalized RQ factorization.

Syntax

```
call sgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call dgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call cgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call zgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call gglse(a, b, c, d, x [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves the linear equality-constrained least squares (LSE) problem:

minimize $||c - A*x||^2$ subject to $B*x = d$

where *A* is an *m*-by-*n* matrix, *B* is a *p*-by-*n* matrix, *c* is a given *m*-vector, and*d* is a given *p*-vector. It is assumed that $p \le n \le m+p$, and

$$\mathrm{rank}(B) = p \quad \text{and} \quad \mathrm{rank}\begin{pmatrix} A \\ B \end{pmatrix} = n .$$

These conditions ensure that the LSE problem has a unique solution, which is obtained using a generalized *RQ* factorization of the matrices (B, A) given by

```
B=(0 R)*Q, A=Z*T*Q
```

Input Parameters

m	INTEGER. The number of rows of the matrix *A* ($m \ge 0$).
n	INTEGER. The number of columns of the matrices *A* and *B* ($n \ge 0$).
p	INTEGER. The number of rows of the matrix *B* ($0 \le p \le n \le m+p$).
a, *b*, *c*, *d*, *work*	REAL for sgglse DOUBLE PRECISION for dgglse COMPLEX for cgglse DOUBLE COMPLEX for zgglse. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *p*-by-*n*matrix *B*.

The second dimension of *b* must be at least max(1, *n*).

c(*), size at least max(1, *m*), contains the right hand side vector for the least squares part of the LSE problem.

d(*),, size at least max(1, *p*), contains the right hand side vector for the constrained equation.

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of *a*; at least max(1, *m*).

ldb — `INTEGER`. The leading dimension of *b*; at least max(1, *p*).

lwork — `INTEGER`. The size of the *work* array;

lwork≥ `max(1, m+n+p)`.

If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a — The elements on and above the diagonal contain the min(*m*, *n*)-by-*n* upper trapezoidal matrix *T* as returned by `?ggrqf`.

x — `REAL` for `sgglse`

The solution of the LSE problem.

b — On exit, the upper right triangle of the subarray `b(1:p, n-p+1:n)` contains the *p*-by-*p* upper triangular matrix *R* as returned by `?ggrqf`.

d — On exit, *d* is destroyed.

c — On exit, the residual sum-of-squares for the solution is given by the sum of squares of elements *n*-*p*+1 to *m* of vector *c*.

work(1) — If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

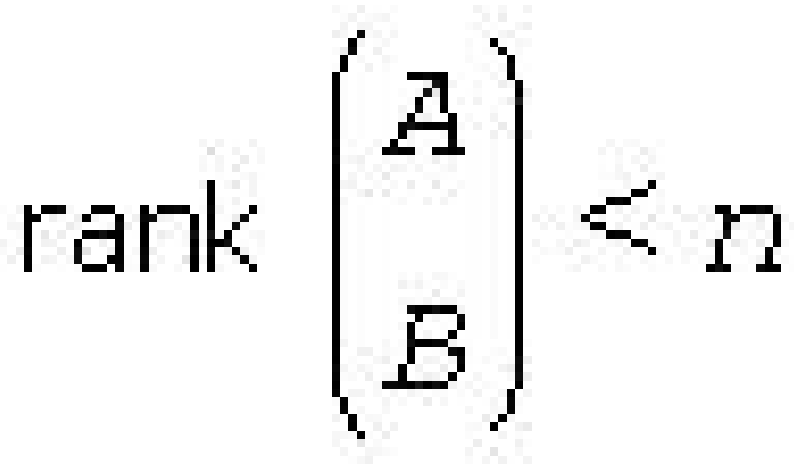
info — `INTEGER`.

If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*-th parameter had an illegal value.

If *info* = 1, the upper triangular factor *R* associated with *B* in the generalized RQ factorization of the pair (*B*, *A*) is singular, so that `rank(B)` < *p*; the least squares solution could not be computed.

If *info* = 2, the (*n*–*p*)-by-(*n*–*p*) part of the upper trapezoidal factor *T* associated with *A* in the generalized RQ factorization of the pair (*B*, *A*) is singular, so that



; the least squares solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gglse` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
b	Holds the matrix *B* of size (*p*,*n*).
c	Holds the vector of length (*m*).
d	Holds the vector of length (*p*).
x	Holds the vector of length *n*.

Application Notes

For optimum performance, use

lwork≥*p* + min(*m*, *n*) + max(*m*, *n*)**nb*,

where *nb* is an upper bound for the optimal blocksizes for `?geqrf`, `?gerqf`, `?ormqr`/`?unmqr` and `?ormrq`/`?unmrq`.

You may set *lwork* to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggglm
Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

Syntax

```
call sggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call dggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call cggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
```

```
call zggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call ggglm(a, b, d, x, y [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves a general Gauss-Markov linear model (GLM) problem:

$\text{minimize}_x \; ||y||_2$ subject to $d = A*x + B*y$

where *A* is an *n*-by-*m* matrix, *B* is an *n*-by-*p* matrix, and *d* is a given *n*-vector. It is assumed that $m \le n \le m+p$, and $\text{rank}(A) = m$ and $\text{rank}(AB) = n$.

Under these assumptions, the constrained equation is always consistent, and there is a unique solution *x* and a minimal 2-norm solution *y*, which is obtained using a generalized *QR* factorization of the matrices (A, B) given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}; \qquad B = Q * T * Z.$$

In particular, if matrix *B* is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

$\text{minimize}_x \; ||B^{-1}(d-A*x)||_2$.

Input Parameters

n	INTEGER. The number of rows of the matrices *A* and *B* ($n \ge 0$).
m	INTEGER. The number of columns in *A* ($m \ge 0$).
p	INTEGER. The number of columns in *B* ($p \ge n - m$).
a, *b*, *d*, *work*	REAL for sggglm DOUBLE PRECISION for dggglm COMPLEX for cggglm DOUBLE COMPLEX for zggglm. Arrays: *a*(*lda*,*) contains the *n*-by-*m* matrix *A*. The second dimension of *a* must be at least max(1, *m*). *b*(*ldb*,*) contains the *n*-by-*p* matrix *B*. The second dimension of *b* must be at least max(1, *p*). *d*(*), size at least max(1, *n*), contains the left hand side of the GLM equation. *work* is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The leading dimension of *b*; at least max(1, *n*).

lwork — INTEGER. The size of the *work* array; *lwork*≥ max(1, *n*+*m*+*p*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

x, *y* — REAL for sggglm

DOUBLE PRECISION for dggglm

COMPLEX for cggglm

DOUBLE COMPLEX for zggglm.

Arrays *x*(*), *y*(*). size at least max(1, *m*) for *x* and at least max(1, *p*) for *y*.

On exit, *x* and *y* are the solutions of the GLM problem.

a — On exit, the upper triangular part of the array *a* contains the *m*-by-*m* upper triangular matrix *R*.

b — On exit, if *n* ≤ *p*, the upper right triangle of the subarray *b*(1:*n*, *p*-*n*+1:*p*) contains the *n*-by-*n* upper triangular matrix *T* as returned by ?ggrqf; if *n* > *p*, the elements on and above the (*n*-*p*)-th subdiagonal contain the *n*-by-*p* upper trapezoidal matrix *T*.

d — On exit, *d* is destroyed

work(1) — If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, the upper triangular factor *R* associated with *A* in the generalized QR factorization of the pair (*A*, *B*) is singular, so that rank(*A*) < *m*; the least squares solution could not be computed.

If *info* = 2, the bottom (*n*-*m*)-by-(*n*-*m*) part of the upper trapezoidal factor *T* associated with *B* in the generalized QR factorization of the pair (*A*, *B*) is singular, so that rank(*AB*) < *n*; the least squares solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ggglm interface are the following:

a	Holds the matrix *A* of size (*n*,*m*).
b	Holds the matrix *B* of size (*n*,*p*).
d	Holds the vector of length *n*.
x	Holds the vector of length (*m*).
y	Holds the vector of length (*p*).

Application Notes

For optimum performance, use

lwork≥*m* + min(*n*, *p*) + max(*n*, *p*)**nb*,

where *nb* is an upper bound for the optimal blocksizes for ?geqrf, ?gerqf, ?ormqr/?unmqr and ?ormrq/?unmrq.

You may set *lwork* to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Symmetric Eigenvalue Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving symmetric eigenvalue problems. See also computational routines that can be called to solve these problems. Table "Driver Routines for Solving Symmetric Eigenproblems" lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Symmetric Eigenproblems

Routine Name	Operation performed
syev/heev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix.
syevd/heevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix using divide and conquer algorithm.
syevx/heevx	Computes selected eigenvalues and, optionally, eigenvectors of a symmetric / Hermitian matrix.
syevr/heevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix using the Relatively Robust Representations.
spev/hpev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
spevd/hpevd	Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix held in packed storage.
spevx/hpevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
sbev /hbev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
sbevd/hbevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian band matrix using divide and conquer algorithm.

Routine Name	Operation performed
sbevx/hbevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
stev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.
stevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.
stevx	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
stevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

?syev
Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix.

Syntax

```
call ssyev(jobz, uplo, n, a, lda, w, work, lwork, info)
call dsyev(jobz, uplo, n, a, lda, w, work, lwork, info)
call syev(a, w [,jobz] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*.

Note that for most cases of real symmetric eigenvalue problems the default choice should be syevr function as its underlying algorithm is faster and uses less workspace.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

a, *work*

REAL for ssyev

DOUBLE PRECISION for dsyev

a(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of the array *a*.

Must be at least max(1, *n*).

lwork — `INTEGER`.

The dimension of the array *work*.

Constraint: *lwork* ≥ `max(1, 3n-1)`.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a — On exit, if *jobz* = 'V', then if *info* = 0, array *a* contains the orthonormal eigenvectors of the matrix *A*.

If *jobz* = 'N', then on exit the lower triangle

(if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w — `REAL` for `ssyev`

`DOUBLE PRECISION` for `dsyev`

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

work(1) — On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

info — `INTEGER`.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `syev` interface are the following:

a — Holds the matrix *A* of size (*n*, *n*).

w — Holds the vector of length *n*.

job	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance set *lwork*≥ (*nb*+2)***n*, where *nb* is the blocksize for ?sytrd returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

If *lwork* has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array *work*. This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?heev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call cheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
call zheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
call heev(a, w [,jobz] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A*.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be heevr function as its underlying algorithm is faster and uses less workspace.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

a, *work*

COMPLEX for cheev

DOUBLE COMPLEX for zheev

Arrays:

a(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*. C

onstraint: $lwork \geq \max(1, 2n-1)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

rwork

REAL for cheev

DOUBLE PRECISION for zheev.

Workspace array, size at least max(1, 3*n*-2).

Output Parameters

a

On exit, if *jobz* = 'V', then if *info* = 0, array *a* contains the orthonormal eigenvectors of the matrix *A*.

If *jobz* = 'N', then on exit the lower triangle

(if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w

REAL for cheev

DOUBLE PRECISION for zheev

Array, size at least max(1, *n*).

	If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.
work(1)	On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine heev interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length *n*.
job	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use

lwork≥ (*nb*+1)**n*,

where *nb* is the blocksize for ?hetrd returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?syevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

Syntax

```
call ssyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
call dsyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
```

```
call syevd(a, w [,jobz] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix *A*. In other words, it can compute the spectral factorization of *A* as: $A = Z*\Lambda*Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and *Z* is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$A*z_i = \lambda_i*z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Note that for most cases of real symmetric eigenvalue problems the default choice should be syevr function as its underlying algorithm is faster and uses less workspace. ?syevd requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

a

REAL for ssyevd

DOUBLE PRECISION for dsyevd

Array, size (*lda*, *).

a(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

lda

INTEGER. The leading dimension of the array *a*.

Must be at least max(1, *n*).

work

REAL for ssyevd

DOUBLE PRECISION for dsyevd.

Workspace array, size at least *lwork*.

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

if $n \leq 1$, then $lwork \geq 1$;

if *jobz* = 'N' and $n > 1$, then $lwork \geq 2*n + 1$;

if *jobz* = 'V' and $n > 1$, then $lwork \geq 2*n^2 + 6*n + 1$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork

INTEGER.

Workspace array, its dimension max(1, *liwork*).

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

if $n \leq 1$, then $liwork \geq 1$;

if *jobz* = 'N' and $n > 1$, then $liwork \geq 1$;

if *jobz* = 'V' and $n > 1$, then $liwork \geq 5*n + 3$.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

w

REAL for ssyevd

DOUBLE PRECISION for dsyevd

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

a

If *jobz* = 'V', then on exit this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*.

work(1)

On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

iwork(1)

On exit, if *liwork* > 0, then *iwork(1)* returns the required minimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, and *jobz* = 'N', then the algorithm failed to converge; *i* indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If *info* = *i*, and *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*,*n*+1).

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine syevd interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length *n*.
jobz	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $||E||_2 = O(\varepsilon)*||A||_2$, where ε is the machine precision.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is heevd

?heevd
Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

Syntax

```
call cheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info)
call zheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info)
call heevd(a, w [,job] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix *A*. In other words, it can compute the spectral factorization of *A* as: $A = Z*\Lambda*Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and *Z* is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$A*z_i = \lambda_i*z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Note that for most cases of complex Hermetian eigenvalue problems the default choice should be heevr function as its underlying algorithm is faster and uses less workspace. `?heevd` requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

jobz

`CHARACTER*1`. Must be `'N'` or `'V'`.

If *jobz* = `'N'`, then only eigenvalues are computed.

If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, *a* stores the upper triangular part of *A*.

If *uplo* = `'L'`, *a* stores the lower triangular part of *A*.

n

`INTEGER`. The order of the matrix *A* ($n \geq 0$).

a

`COMPLEX` for `cheevd`

`DOUBLE COMPLEX` for `zheevd`

Array, size (*lda*, *).

a(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

lda

`INTEGER`. The leading dimension of the array *a*. Must be at least max(1, *n*).

work

`COMPLEX` for `cheevd`

`DOUBLE COMPLEX` for `zheevd`.

Workspace array, size `max(1, lwork)`.

lwork

`INTEGER`.

The dimension of the array *work*. Constraints:

if $n \leq 1$, then $lwork \geq 1$;

if *jobz* = `'N'` and *n* > 1, then $lwork \geq n+1$;

if *jobz* = `'V'` and *n* > 1, then $lwork \geq n^2+2*n$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

rwork

REAL for cheevd

DOUBLE PRECISION for zheevd

Workspace array, size at least *lrwork*.

lrwork

INTEGER.

The dimension of the array *rwork*. Constraints:

if $n \leq 1$, then *lrwork* ≥ 1;

if *job* = 'N' and $n > 1$, then *lrwork* $\geq n$;

if *job* = 'V' and $n > 1$, then *lrwork* $\geq 2*n^2 + 5*n + 1$.

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork

INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork

INTEGER.

The dimension of the array *iwork*. Constraints: if $n \leq 1$, then *liwork* ≥ 1;

if *jobz* = 'N' and $n > 1$, then *liwork* ≥ 1;

if *jobz* = 'V' and $n > 1$, then *liwork* $\geq 5*n+3$.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

w

REAL for cheevd

DOUBLE PRECISION for zheevd

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

a

If *jobz* = 'V', then on exit this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*.

work(1)

On exit, if *lwork* > 0, then the real part of *work*(1) returns the required minimal size of *lwork*.

rwork(1)	On exit, if *lrwork* > 0, then *rwork*(1) returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, and *jobz* = 'N', then the algorithm failed to converge; *i* off-diagonal elements of an intermediate tridiagonal form did not converge to zero; if *info* = *i*, and *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*, *n*+1). If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine heevd interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length (*n*).
jobz	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $||E||_2 = O(\varepsilon)*||A||_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is syevd. See also hpevd for matrices held in packed storage, and hbevd for banded matrices.

?syevx
Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

```
call ssyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
```

```
call dsyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
```

```
call syevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of real symmetric eigenvalue problems the default choice should be syevr function as its underlying algorithm is faster and uses less workspace. ?syevx is faster for a few selected eigenvalues.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range

CHARACTER*1. Must be 'A', 'V', or 'I'.

If *range* = 'A', all eigenvalues will be found.

If *range* = 'V', all eigenvalues in the half-open interval (*vl*, *vu*] will be found.

If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

a, *work*

REAL for ssyevx

DOUBLE PRECISION for dsyevx.

Arrays:

a(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of the array *a*. Must be at least max(1, *n*) .

vl, *vu* — `REAL` for `ssyevx`

`DOUBLE PRECISION` for `dsyevx`.

If `range = 'V'`, the lower and upper bounds of the interval to be searched for eigenvalues; *vl*≤*vu*. Not referenced if `range = 'A'` or `'I'`.

il, *iu* — `INTEGER`.

If `range = 'I'`, the indices of the smallest and largest eigenvalues to be returned.

Constraints: `1 ≤il≤iu≤n`, if `n > 0`;

`il = 1` and `iu = 0`, if `n = 0`.

Not referenced if `range = 'A'` or `'V'`.

abstol — `REAL` for `ssyevx`

`DOUBLE PRECISION` for `dsyevx`.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz — `INTEGER`. The leading dimension of the output array *z*; `ldz≥ 1`.

If `jobz = 'V'`, then `ldz≥ max(1, n)` .

lwork — `INTEGER`.

The dimension of the array *work*.

If `n≤ 1` then `lwork≥ 1`, otherwise `lwork=8*n`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

iwork — `INTEGER`. Workspace array, size at least max(1, 5*n*).

Output Parameters

a — On exit, the lower triangle (if `uplo = 'L'`) or the upper triangle (if `uplo = 'U'`) of *A*, including the diagonal, is overwritten.

m — `INTEGER`. The total number of eigenvalues found;

`0 ≤m≤n`.

If `range = 'A'`, `m = n`, and if `range = 'I'`, `m = iu-il+1`.

w — `REAL` for `ssyevx`

`DOUBLE PRECISION` for `dsyevx`

Array, size at least max(1, *n*). The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z

`REAL` for `ssyevx`

`DOUBLE PRECISION` for `dsyevx`.

Array *z*(*ldz*,*) contains eigenvectors.

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = `'V'`, then if *info* = `0`, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the i-th column of *z* holding the eigenvector associated with *w*(i).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = `'N'`, then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = `'V'`, the exact value of *m* is not known in advance and an upper bound must be used.

work(1)

On exit, if *lwork* > `0`, then *work*(1) returns the required minimal size of *lwork*.

ifail

`INTEGER`.

Array, size at least max(1, *n*).

If *jobz* = `'V'`, then if *info* = `0`, the first *m* elements of *ifail* are zero; if *info* > `0`, then *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = `'V'`, then *ifail* is not referenced.

info

`INTEGER`.

If *info* = `0`, the execution is successful.

If *info* = `-i`, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `syevx` interface are the following:

a

Holds the matrix *A* of size (*n*, *n*).

w

Holds the vector of length *n*.

a

Holds the matrix *A* of size (*m*, *n*).

ifail

Holds the vector of length *n*.

uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

For optimum performance use *lwork*≥ (*nb*+3)*n, where *nb* is the maximum of the blocksize for ?sytrd and ?ormtr returned by ilaenv.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If *lwork* has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array *work*. This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*|| is used as tolerance, where ||*T*|| is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?heevx
Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call cheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, rwork, iwork, ifail, info)
```

```
call zheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, rwork, iwork, ifail, info)
```

```
call heevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of complex Hermetian eigenvalue problems the default choice should be heevr function as its underlying algorithm is faster and uses less workspace. ?heevx is faster for a few selected eigenvalues.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range

CHARACTER*1. Must be 'A', 'V', or 'I'.

If *range* = 'A', all eigenvalues will be found.

If *range* = 'V', all eigenvalues in the half-open interval (*vl*, *vu*] will be found.

If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

a, *work*

COMPLEX for cheevx

DOUBLE COMPLEX for zheevx.

Arrays:

a(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).

vl, *vu*

REAL for cheevx

DOUBLE PRECISION for zheevx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; *vl*≤*vu*. Not referenced if *range* = 'A'or 'I'.

il, *iu*

INTEGER.

If *range* = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints:

1 ≤*il*≤*iu*≤*n*, if *n* > 0;*il* = 1 and *iu* = 0, if *n* = 0. Not referenced if *range* = 'A'or 'V'.

abstol

REAL for cheevx

DOUBLE PRECISION for zheevx. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1.

If *jobz* = 'V', then *ldz*≥max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*.

lwork≥ 1 if *n*≤1; otherwise at least 2**n*.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

rwork

REAL for cheevx

DOUBLE PRECISION for zheevx.

Workspace array, size at least max(1, 7*n*).

iwork

INTEGER. Workspace array, size at least max(1, 5*n*).

Output Parameters

a

On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m

INTEGER. The total number of eigenvalues found; 0 ≤*m*≤*n*.

If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+*1*.

w

REAL for cheevx

DOUBLE PRECISION for zheevx

Array, size max(1, *n*). The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z

COMPLEX for cheevx

DOUBLE COMPLEX for zheevx.

Array *z*(*ldz*,*) contains eigenvectors.

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the i-th column of *z* holding the eigenvector associated with *w*(i).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

work(1) On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

ifail INTEGER.

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, then *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'V', then *ifail* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine heevx interface are the following:

a Holds the matrix *A* of size (*n*, *n*).

w Holds the vector of length *n*.

z Holds the matrix *Z* of size (*n*, *n*).

ifail Holds the vector of length *n*.

uplo Must be 'U' or 'L'. The default value is 'U'.

vl Default value for this element is *vl* = -HUGE(*vl*).

vu Default value for this element is *vu* = HUGE(*vl*).

il Default value for this argument is *il* = 1.

iu Default value for this argument is *iu* = *n*.

abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

For optimum performance use *lwork*≥ (*nb*+1)*n, where *nb* is the maximum of the blocksize for ?hetrd and ?unmtr returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*|| will be used in its place, where ||*T*|| is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?syevr
Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.

Syntax

```
call ssyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
```

```
call dsyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
```

```
call syevr(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix *A* to tridiagonal form *T*. Then, whenever possible, `?syevr` calls stemr to compute the eigenspectrum using Relatively Robust Representations. stemr computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L*D*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the each unreduced block of *T*:

a. Compute $T - \sigma*I = L*D*L^T$, so that *L* and *D* define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of *D* and *L* cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix *T* does not have this property in general.

b. Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).

c. For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.

d. For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?syevr` calls stemr when the full spectrum is requested on machines that conform to the IEEE-754 floating point standard. `?syevr` calls stebz and stein on non-IEEE machines and when partial spectrum requests are made.

Normal execution of `?dsyevr` may create NaNs and infinities and may abort due to a floating point exception in environments that do not handle NaNs and infinities in the IEEE standard default manner.

Note that `?syevr` is preferable for most cases of real symmetric eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

jobz

`CHARACTER*1`. Must be `'N'` or `'V'`.

If *jobz* = `'N'`, then only eigenvalues are computed.

If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

range

`CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.

If *range* = `'A'`, the routine computes all eigenvalues.

If *range* = `'V'`, the routine computes eigenvalues *w*(*i*) in the half-open interval:

vl < *w*(*i*)≤*vu*.

If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

For *range* = `'V'`or `'I'` and *iu*-*il* < *n*-1, sstebz/dstebz and sstein/dstein are called.

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

a, *work* — REAL for ssyevr

DOUBLE PRECISION for dsyevr.

Arrays:

a(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).

vl, *vu* — REAL for ssyevr

DOUBLE PRECISION for dsyevr.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl* < *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* — INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint:

1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0;

il = 1 and *iu* = 0, if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for ssyevr

DOUBLE PRECISION for dsyevr. The absolute error tolerance to which each eigenvalue/eigenvector is required.

If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*.

If *abstol* < *n* *eps*||*T*||, then *n* *eps*||*T*|| is used instead, where eps is the machine precision, and ||*T*|| is the 1-norm of the matrix *T*. The eigenvalues are computed to an accuracy of eps*||*T*|| irrespective of *abstol*.

If high relative accuracy is important, set *abstol* to ?lamch('S').

ldz — INTEGER. The leading dimension of the output array *z*.

Constraints:

ldz ≥ 1 if *jobz* = 'N' and

ldz ≥ max(1, *n*) if *jobz* = 'V'.

lwork INTEGER.

The dimension of the array *work*.

Constraint: *lwork* ≥ max(1, 26*n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.

The dimension of the array *iwork*, *lwork* ≥ max(1, 10*n*).

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*.

If *range* = 'A', *m* = *n*, if *range* = 'I', *m* = *iu*-*il*+1, and if *range* = 'V' the exact value of *m* is not known in advance.

w, *z* REAL for ssyevr

DOUBLE PRECISION for dsyevr.

Arrays:

w(*), size at least max(1, *n*), contains the selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*);

z(*ldz*,*), the second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If *jobz* = 'N', then *z* is not referenced. Note that you must ensure that at least max(1, *m*) columns are supplied in the array *z* ; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

isuppz INTEGER.

Array, size at least 2 *max(1, *m*).

	The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*). Referenced only if eigenvectors are needed (*jobz* = 'V') and all eigenvalues are needed, that is, *range* = 'A' or *range* = 'I' and *il* = 1 and *iu* = *n*.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, an internal error has occurred.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine syevr interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length *n*.
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
isuppz	Holds the vector of length (2**m*), where the values (2**m*) are significant.
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *isuppz* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

For optimum performance use *lwork*≥ (*nb*+6)**n*, where *nb* is the maximum of the blocksize for ?sytrd and ?ormtr returned by ilaenv.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?heevr
Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using the Relatively Robust Representations.

Syntax

call cheevr(*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call zheevr(*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call heevr(*a*, *w* [,*uplo*] [,*z*] [,*vl*] [,*vu*] [,*il*] [,*iu*] [,*m*] [,*isuppz*] [,*abstol*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix *A* to tridiagonal form *T* with a call to hetrd. Then, whenever possible, ?heevr calls stegr to compute the eigenspectrum using Relatively Robust Representations. ?stegr computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L*D*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For each unreduced block (submatrix) of *T*:

a. Compute $T - \sigma*I = L*D*L^T$, so that *L* and *D* define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of *D* and *L* cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix *T* does not have this property in general.

b. Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).

c. For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.

d. For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?heevr` calls stemr when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard, or stebz and stein on non-IEEE machines and when partial spectrum requests are made.

Note that the routine `?heevr` is preferable for most cases of complex Hermitian eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

jobz

`CHARACTER*1`. Must be `'N'` or `'V'`.

If *job* = `'N'`, then only eigenvalues are computed.

If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

range

`CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.

If *range* = `'A'`, the routine computes all eigenvalues.

If *range* = `'V'`, the routine computes eigenvalues *lambda*`(i)` in the half-open interval: *vl*< *lambda*`(i)`≤*vu*.

If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

For *range* = `'V'`or `'I'`, `sstebz`/`dstebz` and `cstein`/`zstein` are called.

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, *a* stores the upper triangular part of *A*.

If *uplo* = `'L'`, *a* stores the lower triangular part of *A*.

n

`INTEGER`. The order of the matrix *A* (*n*≥ `0`).

a, *work*

`COMPLEX` for `cheevr`

`DOUBLE COMPLEX` for `zheevr`.

Arrays:

a(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

lda

`INTEGER`. The leading dimension of the array *a*.

Must be at least max(1, *n*).

vl, *vu*

`REAL` for `cheevr`

`DOUBLE PRECISION` for `zheevr`.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl*< *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for cheevr

DOUBLE PRECISION for zheevr.

The absolute error tolerance to which each eigenvalue/eigenvector is required.

If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*.

If *abstol* < *n* *eps*||*T*||, then *n* *eps*||*T*|| is used instead, where eps is the machine precision, and ||*T*|| is the 1-norm of the matrix *T*. The eigenvalues are computed to an accuracy of eps*||*T*|| irrespective of *abstol*.

If high relative accuracy is important, set *abstol* to ?lamch('S').

ldz

INTEGER. The leading dimension of the output array *z*. Constraints:

ldz≥ 1 if *jobz* = 'N';

ldz≥ max(1, *n*) if *jobz* = 'V'.

lwork

INTEGER.

The dimension of the array *work*.

Constraint: *lwork*≥ max(1, 2*n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

rwork

REAL for cheevr

DOUBLE PRECISION for zheevr.

Workspace array, size max(1, *lwork*).

lrwork

INTEGER.

The dimension of the array *rwork*;

lwork≥ max(1, 24*n*).

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.

The dimension of the array *iwork*,

lwork≥ max(1, 10*n*).

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m INTEGER. The total number of eigenvalues found,

0 ≤*m*≤*n*.

If *range* = 'A', *m* = *n*, if *range* = 'I', *m* = *iu*-*il*+1, and if *range* = 'V' the exact value of *m* is not known in advance.

w REAL for cheevr

DOUBLE PRECISION for zheevr.

Array, size at least max(1, *n*), contains the selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*).

z COMPLEX for cheevr

DOUBLE COMPLEX for zheevr.

Array *z*(*ldz*,*), the second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

isuppz INTEGER.

Array, size at least 2 *max(1, *m*).

	The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*). Referenced only if eigenvectors are needed (*jobz* = 'V') and all eigenvalues are needed, that is, *range* = 'A' or *range* = 'I' and *il* = 1 and *iu* = *n*.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
rwork(1)	On exit, if *info* = 0, then *rwork(1)* returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, an internal error has occurred.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine heevr interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length *n*.
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
isuppz	Holds the vector of length (2**n*), where the values (2**m*) are significant.
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *isuppz* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

For optimum performance use *lwork*≥ (*nb*+1)***n*, where *nb* is the maximum of the blocksize for ?hetrd and ?unmtr returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *lrwork*, or *liwork*) for the first run or set *lwork* = -1 (*lrwork* = -1, *liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *lrwork*, *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *rwork*, *iwork*) on exit. Use this value (*work*(1), *rwork*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *rwork*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*lrwork*, *liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Normal execution of ?stemr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

For more details, see ?stemr and these references:

- Inderjit S. Dhillon and Beresford N. Parlett: "Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices," Linear Algebra and its Applications, 387(1), pp. 1-28, August 2004.
- Inderjit Dhillon and Beresford Parlett: "Orthogonal Eigenvectors and Relative Gaps," SIAM Journal on Matrix Analysis and Applications, Vol. 25, 2004. Also LAPACK Working Note 154.
- Inderjit Dhillon: "A new O(n^2) algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.

?spev
Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

```
call sspev(jobz, uplo, n, ap, w, z, ldz, work, info)
call dspev(jobz, uplo, n, ap, w, z, ldz, work, info)
call spev(ap, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.

If *job* = 'N', then only eigenvalues are computed.

If *job* = 'V', then eigenvalues and eigenvectors are computed.

uplo — CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*.

If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

ap, *work* — REAL for sspev

DOUBLE PRECISION for dspev

Arrays:

Array *ap*(*) contains the packed upper or lower triangle of symmetric matrix *A*, as specified by *uplo*.

The size of *ap* must be at least max(1, *n**(*n*+1)/2).

work(*) is a workspace array, size at least max(1, 3*n*).

ldz — INTEGER. The leading dimension of the output array *z*. Constraints:

if *jobz* = 'N', then *ldz* ≥ 1;

if *jobz* = 'V', then *ldz* ≥ max(1, *n*).

Output Parameters

w, *z* — REAL for sspev

DOUBLE PRECISION for dspev

Arrays:

w(*), size at least max(1, *n*).

If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order.

z(*ldz*,*). The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If *jobz* = 'N', then *z* is not referenced.

ap — On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `spev` interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted.

?hpev
Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

```
call chpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call zhpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call hpev(ap, w [,uplo] [,z] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* in packed storage.

Input Parameters

jobz	`CHARACTER*1`. Must be `'N'` or `'V'`. If *job* = `'N'`, then only eigenvalues are computed. If *job* = `'V'`, then eigenvalues and eigenvectors are computed.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, *ap* stores the packed upper triangular part of *A*. If *uplo* = `'L'`, *ap* stores the packed lower triangular part of *A*.
n	`INTEGER`. The order of the matrix *A* (*n* ≥ 0).
ap	`COMPLEX` for `chpev` `DOUBLE COMPLEX` for `zhpev`.

Array *ap*(*) contains the packed upper or lower triangle of Hermitian matrix A, as specified by *uplo*.

The size of *ap* must be at least max(1, *n**(*n*+1)/2).

work

COMPLEX for chpev

DOUBLE COMPLEX for zhpev.

(*) is a workspace array, size at least max(1, 2*n*-1).

ldz

INTEGER. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then *ldz*≥ 1;

if *jobz* = 'V', then *ldz*≥ max(1, *n*) .

rwork

REAL for chpev

DOUBLE PRECISION for zhpev.

Workspace array, size at least max(1, 3*n*-2).

Output Parameters

w

REAL for chpev

DOUBLE PRECISION for zhpev.

Array, size at least max(1, *n*).

If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order.

z

COMPLEX for chpev

DOUBLE COMPLEX for zhpev.

Array z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If *jobz* = 'N', then *z* is not referenced.

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hpev` interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted.

?spevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix held in packed storage.

Syntax

```
call sspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
call dspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
call spevd(ap, w [,uplo] [,z] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix *A* (held in packed storage). In other words, it can compute the spectral factorization of *A* as:

$A = Z*\Lambda*Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and *Z* is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$A*z_i = \lambda_i*z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

jobz	`CHARACTER*1`. Must be `'N'` or `'V'`. If *jobz* = `'N'`, then only eigenvalues are computed. If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*.

If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

ap, *work*

REAL for sspevd

DOUBLE PRECISION for dspevd

Arrays:

ap(*) contains the packed upper or lower triangle of symmetric matrix *A*, as specified by *uplo*.

The dimension of *ap* must be max(1, $n*(n+1)/2$)

work is a workspace array, its dimension max(1, *lwork*).

ldz

INTEGER. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then $ldz \geq 1$;

if *jobz* = 'V', then *ldz* ≥ max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

if $n \leq 1$, then $lwork \geq 1$;

if *jobz* = 'N' and $n > 1$, then $lwork \geq 2*n$;

if *jobz* = 'V' and $n > 1$, then

$lwork \geq n^2 + 6*n + 1$.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork

INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

if $n \leq 1$, then $liwork \geq 1$;

if *jobz* = 'N' and $n > 1$, then $liwork \geq 1$;

if *jobz* = 'V' and $n > 1$, then $liwork \geq 5*n+3$.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

w, *z*

REAL for sspevd

DOUBLE PRECISION for dspevd

Arrays:

w(*), size at least max(1, *n*).

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

z(*ldz*,*).

The second dimension of *z* must be: at least 1 if *jobz* = 'N';at least max(1, *n*) if *jobz* = 'V'.

If *jobz* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. If *jobz* = 'N', then *z* is not referenced.

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required *lwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the required *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine spevd interface are the following:

ap

Holds the array *A* of size (*n**(*n*+1)/2).

w

Holds the vector with the number of elements *n*.

z

Holds the matrix *Z* of size (*n*, *n*).

uplo

Must be 'U' or 'L'. The default value is 'U'.

jobz Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $||E||_2 = O(\varepsilon)*||A||_2$, where ε is the machine precision.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is hpevd.

See also syevd for matrices held in full storage, and sbevd for banded matrices.

?hpevd

Uses divide and conquer algorithm to compute all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix held in packed storage.

Syntax

```
call chpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zhpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call hpevd(ap, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix *A* (held in packed storage). In other words, it can compute the spectral factorization of *A* as: $A = Z*\Lambda*Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and *Z* is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$A*z_i = \lambda_i*z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*.

If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

ap, *work*

COMPLEX for chpevd

DOUBLE COMPLEX for zhpevd

Arrays:

ap(*) contains the packed upper or lower triangle of Hermitian matrix *A*, as specified by *uplo*.

The dimension of *ap* must be at least max(1, $n*(n+1)/2$).

work is a workspace array, its dimension max(1, *lwork*).

ldz

INTEGER. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then $ldz \geq 1$;

if *jobz* = 'V', then $ldz \geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

if $n \leq 1$, then $lwork \geq 1$;

if *jobz* = 'N' and $n > 1$, then $lwork \geq n$;

if *jobz* = 'V' and $n > 1$, then $lwork \geq 2*n$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

rwork

REAL for chpevd

DOUBLE PRECISION for zhpevd

Workspace array, its dimension max(1, *lrwork*).

lrwork

INTEGER.

The dimension of the array *rwork*. Constraints:

if $n \leq 1$, then $lrwork \geq 1$;

if *jobz* = 'N' and $n > 1$, then $lrwork \geq n$;

if *jobz* = 'V' and *n* > 1, then *lrwork* ≥ 2*n^2 + 5*n + 1.

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.

The dimension of the array *iwork*.

Constraints:

if *n* ≤ 1, then *liwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *liwork* ≥ 1;

if *jobz* = 'V' and *n* > 1, then *liwork* ≥ 5*n+3.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

w REAL for chpevd

DOUBLE PRECISION for zhpevd

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

z COMPLEX for chpevd

DOUBLE COMPLEX for zhpevd

Array, size (*ldz*,*).

The second dimension of *z* must be:

at least 1 if *jobz* = 'N';

at least max(1, *n*) if *jobz* = 'V'.

If *jobz* = 'V', then this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*.

If *jobz* = 'N', then *z* is not referenced.

ap On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.

work(1) On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork(1)	On exit, if *info* = 0, then *rwork*(1) returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hpevd interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $||E||_2 = O(\varepsilon)*||A||_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is spevd.

See also heevd for matrices held in full storage, and hbevd for banded matrices.

?spevx
Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

```
call sspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call dspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call spevx(ap, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *job* = 'N', then only eigenvalues are computed. If *job* = 'V', then eigenvalues and eigenvectors are computed.
range	CHARACTER*1. Must be 'A' or 'V' or 'I'. If *range* = 'A', the routine computes all eigenvalues. If *range* = 'V', the routine computes eigenvalues *w(i)* in the half-open interval: $vl < w(i) \le vu$. If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*. If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* ($n \ge 0$).
ap, *work*	REAL for sspevx DOUBLE PRECISION for dspevx Arrays: Array *ap*(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*. The size of *ap* must be at least max(1, *n**(*n*+1)/2). *work(*)* is a workspace array, size at least max(1, 8*n*).
vl, *vu*	REAL for sspevx

DOUBLE PRECISION for dspevx

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl* < *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0

if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for sspevx

DOUBLE PRECISION for dspevx

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldz

INTEGER. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then *ldz*≥ 1;

if *jobz* = 'V', then *ldz*≥ max(1, *n*).

iwork

INTEGER. Workspace array, size at least max(1, 5*n*).

Output Parameters

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

m

INTEGER. The total number of eigenvalues found,

0 ≤*m*≤*n*. If *range* = 'A', *m* = *n*, if *range* = 'I', *m* = *iu*-*il*+1, and if *range* = 'V' the exact value of *m* is not known in advance..

w, *z*

REAL for sspevx

DOUBLE PRECISION for dspevx

Arrays:

w(*), size at least max(1, *n*).

If *info* = 0, contains the selected eigenvalues of the matrix *A* in ascending order.

z(*ldz*,*).

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail INTEGER.

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine spevx interface are the following:

ap Holds the array *A* of size (*n**(*n*+1)/2).

w Holds the vector with the number of elements *n*.

z Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.

ifail Holds the vector with the number of elements *n*.

uplo Must be 'U' or 'L'. The default value is 'U'.

vl Default value for this element is *vl* = -HUGE(*vl*).

vu Default value for this element is *vu* = HUGE(*vl*).

il Default value for this argument is *il* = 1.

iu Default value for this argument is *iu* = *n*.

abstol Default value for this element is *abstol* = 0.0_WP.

jobz Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted

Note that there will be an error condition if *ifail* is present and *z* is omitted.

range Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:

range = 'V', if one of or both *vl* and *vu* are present,

range = 'I', if one of or both *il* and *iu* are present,

range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hpevx
Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

```
call chpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call zhpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call hpevx(ap, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.

If *job* = 'N', then only eigenvalues are computed.

If *job* = 'V', then eigenvalues and eigenvectors are computed.

range CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *w(i)* in the half-open interval: *vl*< *w(i)*≤*vu*.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*.

If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.

n INTEGER. The order of the matrix *A* (*n*≥ 0).

ap, *work* COMPLEX for chpevx

DOUBLE COMPLEX for zhpevx

Arrays:

Array *ap*(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.

The size of *ap* must be at least max(1, *n**(*n*+1)/2).

work(*) is a workspace array, size at least max(1, 2*n*).

vl, *vu* REAL for chpevx

DOUBLE PRECISION for zhpevx

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl*< *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol REAL for chpevx

DOUBLE PRECISION for zhpevx

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldz INTEGER. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then *ldz*≥ 1;

if *jobz* = 'V', then *ldz*≥ max(1, *n*).

rwork REAL for chpevx

DOUBLE PRECISION for zhpevx

Workspace array, size at least max(1, 7*n*).

iwork INTEGER. Workspace array, size at least max(1, 5*n*).

Output Parameters

ap On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

m INTEGER. The total number of eigenvalues found, $0 \le m \le n$.

$0 \le m \le n$. If *range* = 'A', *m* = *n*, if *range* = 'I', *m* = *iu*-*il*+1, and if *range* = 'V' the exact value of *m* is not known in advance..

w REAL for chpevx

DOUBLE PRECISION for zhpevx

Array, size at least max(1, *n*).

If *info* = 0, contains the selected eigenvalues of the matrix *A* in ascending order.

z COMPLEX for chpevx

DOUBLE COMPLEX for zhpevx

Array z(*ldz*,*).

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail INTEGER.

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hpevx interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector with the number of elements *n*.
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = –HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?sbev
Computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

```
call ssbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
call dsbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
call sbev(ab, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*.

Input Parameters

jobz
CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then only eigenvalues are computed.
If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo
CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *ab* stores the upper triangular part of *A*.
If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n
INTEGER. The order of the matrix *A* (*n*≥ 0).

kd
INTEGER. The number of super- or sub-diagonals in *A*
(*kd*≥ 0).

ab, *work*
REAL for ssbev
DOUBLE PRECISION for dsbev.
Arrays:
ab(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.
The second dimension of *ab* must be at least max(1, *n*).
work (*) is a workspace array.
The dimension of *work* must be at least max(1, 3*n*-2).

ldab
INTEGER. The leading dimension of *ab*; must be at least *kd* +1.

ldz
INTEGER. The leading dimension of the output array *z*.
Constraints:
if *jobz* = 'N', then *ldz*≥ 1;
if *jobz* = 'V', then *ldz*≥ max(1, *n*) .

Output Parameters

w, *z*	REAL for ssbev DOUBLE PRECISION for dsbev Arrays: *w*(*), size at least max(1, *n*). If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. z(*ldz*,*). The second dimension of *z* must be at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If *jobz* = 'N', then *z* is not referenced.
ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form (see the description of ?sbtrd).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sbev interface are the following:

ab	Holds the array *A* of size (*kd*+1,*n*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

?hbev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

```
call chbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
```

```
call zhbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
call hbev(ab, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix *A*.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ab* stores the upper triangular part of *A*.

If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* (*n*≥ 0).

kd

INTEGER. The number of super- or sub-diagonals in *A*

(*kd*≥ 0).

ab, *work*

COMPLEX for chbev

DOUBLE COMPLEX for zhbev.

Arrays:

ab(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of *ab* must be at least max(1, *n*).

work (*) is a workspace array.

The dimension of *work* must be at least max(1, *n*).

ldab

INTEGER. The leading dimension of *ab*; must be at least *kd* +1.

ldz

INTEGER. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then *ldz*≥ 1;

if *jobz* = 'V', then *ldz*≥ max(1, *n*) .

rwork

REAL for chbev

DOUBLE PRECISION for zhbev

Workspace array, size at least max(1, 3*n*-2).

Output Parameters

w	`REAL` for `chbev`
	`DOUBLE PRECISION` for `zhbev`
	Array, size at least max(1, *n*).
	If *info* = 0, contains the eigenvalues in ascending order.

z	`COMPLEX` for `chbev` `DOUBLE COMPLEX` for `zhbev`. Array `z`(*ldz*,*). The second dimension of *z* must be at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If *jobz* = 'N', then *z* is not referenced.
ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form(see the description of hbtrd).
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hbev` interface are the following:

ab	Holds the array *A* of size (*kd+1*,*n*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

?sbevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric band matrix using divide and conquer algorithm.

Syntax

call ssbevd(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call dsbevd(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call sbevd(*ab*, *w* [,*uplo*] [,*z*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric band matrix *A*. In other words, it can compute the spectral factorization of *A* as:

$A = Z*\Lambda*Z^T$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and *Z* is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$A*z_i = \lambda_i*z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of *A*. If *uplo* = 'L', *ab* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n*≥ 0).
kd	INTEGER. The number of super- or sub-diagonals in *A* (*kd*≥ 0).
ab, *work*	REAL for ssbevd DOUBLE PRECISION for dsbevd. Arrays: *ab*(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
ldab	INTEGER. The leading dimension of *ab*; must be at least *kd*+1.
ldz	INTEGER. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then *ldz* ≥ 1;

if *jobz* = 'V', then *ldz* ≥ max(1, *n*) .

lwork INTEGER.

The dimension of the array *work*.

Constraints:

if *n* ≤ 1, then *lwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *lwork* ≥ 2*n*;

if *jobz* = 'V' and *n* > 1, then *lwork* ≥ $2*n^2 + 5*n + 1$.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.

The dimension of the array *iwork*. Constraints: if *n* ≤ 1, then *liwork* < 1; if *job* = 'N' and *n* > 1, then *liwork* < 1; if *job* = 'V' and *n* > 1, then *liwork* < 5*n*+3.

If *liwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

w, *z* REAL for ssbevd

DOUBLE PRECISION for dsbevd

Arrays:

w(*), size at least max(1, *n*).

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

z(*ldz*,*).

The second dimension of *z* must be:

at least 1 if *job* = 'N';

at least max(1, *n*) if *job* = 'V'.

If *job* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. The *i*-th column of *Z* contains the eigenvector which corresponds to the eigenvalue *w*(*i*).

If *job* = 'N', then *z* is not referenced.

ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
work(1)	On exit, if *lwork* > 0, then *work*(1) returns the required minimal size of *lwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sbevd interface are the following:

ab	Holds the array *A* of size (*kd*+1,*n*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $||E||_2=O(\varepsilon)*||A||_2$, where ε is the machine precision.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If any of admissible *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *work* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is hbevd.

See also syevd for matrices held in full storage, and spevd for matrices held in packed storage.

?hbevd
Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian band matrix using divide and conquer algorithm.

Syntax

`call chbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)`

`call zhbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)`

`call hbevd(ab, w [,uplo] [,z] [,info])`

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian band matrix *A*. In other words, it can compute the spectral factorization of *A* as: $A = Z*\Lambda*Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and *Z* is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$A*z_i = \lambda_i*z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

jobz	`CHARACTER*1`. Must be `'N'` or `'V'`. If *jobz* = `'N'`, then only eigenvalues are computed. If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, *ab* stores the upper triangular part of *A*. If *uplo* = `'L'`, *ab* stores the lower triangular part of *A*.
n	`INTEGER`. The order of the matrix *A* (*n* ≥ 0).
kd	`INTEGER`. The number of super- or sub-diagonals in *A* (*kd* ≥ 0).
ab, *work*	`COMPLEX` for `chbevd` `DOUBLE COMPLEX` for `zhbevd`. Arrays: *ab*(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*).

work (*) is a workspace array, its dimension `max(1, lwork)`.

ldab `INTEGER`. The leading dimension of *ab*; must be at least *kd*+1.

ldz `INTEGER`. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then *ldz* ≥ 1;

if *jobz* = 'V', then *ldz* ≥ max(1, *n*) .

lwork `INTEGER`.

The dimension of the array *work*.

Constraints:

if *n* ≤ 1, then *lwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *lwork* ≥ *n*;

if *jobz* = 'V' and *n* > 1, then *lwork* ≥ $2*n^2$.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

rwork `REAL` for `chbevd`

`DOUBLE PRECISION` for `zhbevd`

Workspace array, size at least *lrwork*.

lrwork `INTEGER`.

The dimension of the array *rwork*.

Constraints:

if *n* ≤ 1, then *lrwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *lrwork* ≥ *n*;

if *jobz* = 'V' and *n* > 1, then *lrwork* ≥ $2*n^2 + 5*n + 1$.

If *lrwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork `INTEGER`. Workspace array, size `max(1, liwork)`.

liwork `INTEGER`.

The dimension of the array *iwork*.

Constraints:

if *jobz* = 'N' or *n* ≤ 1, then *liwork* ≥ 1;

if *jobz* = 'V' and *n* > 1, then *liwork* ≥ 5*n+3.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

w

REAL for chbevd

DOUBLE PRECISION for zhbevd

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

z

COMPLEX for chbevd

DOUBLE COMPLEX for zhbevd

Array, size (*ldz*,*).

The second dimension of *z* must be:

at least 1 if *jobz* = 'N';

at least max(1, *n*) if *jobz* = 'V'.

If *jobz* = 'V', then this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*. The *i*-th column of *Z* contains the eigenvector which corresponds to the eigenvalue *w*(*i*).

If *jobz* = 'N', then *z* is not referenced.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

work(1)

On exit, if *lwork* > 0, then the real part of *work*(1) returns the required minimal size of *lwork*.

rwork(1)

On exit, if *lrwork* > 0, then *rwork*(1) returns the required minimal size of *lrwork*.

iwork(1)

On exit, if *liwork* > 0, then *iwork*(1) returns the required minimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hbevd` interface are the following:

ab	Holds the array *A* of size (*kd+1,n*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $||E||_2 = O(\varepsilon)||A||_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set $lwork = -1$ ($liwork = -1$, $lrwork = -1$).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set $lwork = -1$ ($liwork = -1$, $lrwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is sbevd.

See also heevd for matrices held in full storage, and hpevd for matrices held in packed storage.

?sbevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

```
call ssbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call dsbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call sbevx(ab, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range

CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *w(i)* in the half-open interval: $vl < w(i) \le vu$.

If *range* = 'I', the routine computes eigenvalues with indices in range *il* to *iu*.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ab* stores the upper triangular part of *A*.

If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A* ($n \ge 0$).

kd

INTEGER. The number of super- or sub-diagonals in *A*

($kd \ge 0$).

ab, *work*

REAL for ssbevx

DOUBLE PRECISION for dsbevx.

Arrays:

Arrays:

Array *ab*(*lda*,*) contains either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of *ab* must be at least max(1, *n*).

work (*) is a workspace array.

The dimension of *work* must be at least max(1, 7*n*).

ldab

INTEGER. The leading dimension of *ab*; must be at least *kd* +1.

vl, *vu*

REAL for ssbevx

DOUBLE PRECISION for dsbevx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \le il \le iu \le n$, if $n > 0$; $il=1$ and $iu=0$

if $n = 0$.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for chpevx

DOUBLE PRECISION for zhpevx

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldq, *ldz*

INTEGER. The leading dimensions of the output arrays *q* and *z*, respectively.

Constraints:

$ldq \ge 1$, $ldz \ge 1$;

If *jobz* = 'V', then $ldq \ge \max(1, n)$ and $ldz \ge \max(1, n)$.

iwork

INTEGER. Workspace array, size at least max(1, 5*n*).

Output Parameters

q

REAL for ssbevxDOUBLE PRECISION for dsbevx.

Array, size (*ldz*,*n*).

If *jobz* = 'V', the *n*-by-*n* orthogonal matrix is used in the reduction to tridiagonal form.

If *jobz* = 'N', the array *q* is not referenced.

m

INTEGER. The total number of eigenvalues found, $0 \le m \le n$.

If *range* = 'A', $m = n$, if *range* = 'I', $m = iu-il+1$, and if *range* = 'V', the exact value of *m* is not known in advance.

w, *z*

REAL for ssbevx

DOUBLE PRECISION for dsbevx

Arrays:

w(*), size at least max(1, *n*). The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.

z(*ldz*,*).

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if `range = 'V'`, the exact value of *m* is not known in advance and an upper bound must be used.

ab — On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

If `uplo = 'U'`, the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if `uplo = 'L'`, the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

ifail — `INTEGER`.

Array, size at least max(1, *n*).

If `jobz = 'V'`, then if `info = 0`, the first *m* elements of *ifail* are zero; if `info > 0`, the *ifail* contains the indices the eigenvectors that failed to converge.

If `jobz = 'N'`, then *ifail* is not referenced.

info — `INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sbevx` interface are the following:

ab — Holds the array *A* of size (*kd*+*1*,*n*).

w — Holds the vector with the number of elements *n*.

z — Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.

ifail — Holds the vector with the number of elements *n*.

q — Holds the matrix *Q* of size (*n*, *n*).

uplo — Must be `'U'` or `'L'`. The default value is `'U'`.

vl — Default value for this element is *vl* = `-HUGE`(*vl*).

vu — Default value for this element is *vu* = `HUGE`(*vl*).

il — Default value for this argument is `il = 1`.

iu — Default value for this argument is `iu = n`.

abstol — Default value for this element is `abstol = 0.0_WP`.

jobz — Restored based on the presence of the argument *z* as follows:

`jobz = 'V'`, if *z* is present,

jobz = 'N', if *z* is omitted

Note that there will be an error condition if either *ifail* or *q* is present and *z* is omitted.

range Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:

range = 'V', if one of or both *vl* and *vu* are present,

range = 'I', if one of or both *il* and *iu* are present,

range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hbevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

```
call chbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call zhbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call hbevx(ab, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.

If *job* = 'N', then only eigenvalues are computed.

If *job* = 'V', then eigenvalues and eigenvectors are computed.

range CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *w*(*i*) in the half-open interval: *vl* < *w*(*i*) ≤ *vu*.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ab* stores the upper triangular part of *A*.

If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n INTEGER. The order of the matrix *A* (*n* ≥ 0).

kd INTEGER. The number of super- or sub-diagonals in *A*

(*kd* ≥ 0).

ab, *work* COMPLEX for chbevx

DOUBLE COMPLEX for zhbevx.

Arrays:

ab(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of *ab* must be at least max(1, *n*).

work (*) is a workspace array.

The dimension of *work* must be at least max(1, *n*).

ldab INTEGER. The leading dimension of *ab*; must be at least *kd* +1.

vl, *vu* REAL for chbevx

DOUBLE PRECISION for zhbevx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl* < *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol REAL for chbevx

DOUBLE PRECISION for zhbevx.

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldq, *ldz* INTEGER. The leading dimensions of the output arrays *q* and *z*, respectively.

Constraints:

ldq≥ 1, *ldz*≥ 1;

If *jobz* = 'V', then *ldq*≥ max(1, *n*) and *ldz*≥ max(1, *n*).

rwork

REAL for chbevx

DOUBLE PRECISION for zhbevx

Workspace array, size at least max(1, 7*n*).

iwork

INTEGER. Workspace array, size at least max(1, 5*n*).

Output Parameters

q

COMPLEX for chbevxDOUBLE COMPLEX for zhbevx.

Array, size (*ldz*,*n*).

If *jobz* = 'V', the *n*-by-*n* unitary matrix is used in the reduction to tridiagonal form.

If *jobz* = 'N', the array *q* is not referenced.

m

INTEGER. The total number of eigenvalues found,

0 ≤*m*≤*n*.

If *range* = 'A', *m* = *n*, if *range* = 'I', *m* = *iu*-*il*+1, and if *range* = 'V', the exact value of *m* is not known in advance..

w

REAL for chbevx

DOUBLE PRECISION for zhbevx

Array, size at least max(1, *n*). The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z

COMPLEX for chbevx

DOUBLE COMPLEX for zhbevx.

Array z(*ldz*,*).

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

ifail INTEGER.

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hbevx interface are the following:

ab Holds the array *A* of size (*kd*+1,*n*).

w Holds the vector with the number of elements *n*.

z Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.

ifail Holds the vector with the number of elements *n*.

q Holds the matrix *Q* of size (*n*, *n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

vl Default value for this element is *vl* = –HUGE(*vl*).

vu Default value for this element is *vu* = HUGE(*vl*).

il Default value for this argument is *il* = 1.

iu Default value for this argument is *iu* = *n*.

abstol Default value for this element is *abstol* = 0.0_WP.

jobz Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted

Note that there will be an error condition if either *ifail* or *q* is present and *z* is omitted.

range Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:

range = 'V', if one of or both *vl* and *vu* are present,

range = 'I', if one of or both *il* and *iu* are present,

range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon*||T||_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?stev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstev(jobz, n, d, e, z, ldz, work, info)
call dstev(jobz, n, d, e, z, ldz, work, info)
call stev(d, e [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *A*.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

n

INTEGER. The order of the matrix *A* ($n \geq 0$).

d, *e*, *work*

REAL for sstev

DOUBLE PRECISION for dstev.

Arrays:

Array *d*(*) contains the *n* diagonal elements of the tridiagonal matrix *A*.

The size of *d* must be at least max(1, *n*).

Array `e(*)` contains the *n*-1 subdiagonal elements of the tridiagonal matrix *A*.

The size of *e* must be at least max(1, *n*). The *n*-th element of this array is used as workspace.

work(*) is a workspace array.

The dimension of *work* must be at least max(1, 2*n*-2).

If *jobz* = 'N', *work* is not referenced.

ldz — INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1. If *jobz* = 'V' then *ldz*≥ max(1, *n*).

Output Parameters

d — On exit, if *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

z — REAL for sstev

DOUBLE PRECISION for dstev

Array, size (*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with the eigenvalue returned in *d*(*i*).

If *job* = 'N', then *z* is not referenced.

e — On exit, this array is overwritten with intermediate results.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm failed to converge;

i elements of *e* did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine stev interface are the following:

d — Holds the vector of length *n*.

e — Holds the vector of length *n*.

z — Holds the matrix *Z* of size (*n*, *n*).

jobz — Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

?stevd
Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.

Syntax

```
call sstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call stevd(d, e [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix *T*. In other words, the routine can compute the spectral factorization of *T* as: $T = Z*\Lambda*Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and *Z* is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$T*z_i = \lambda_i*z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

There is no complex analogue of this routine.

Input Parameters

jobz
CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then only eigenvalues are computed.
If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

n
INTEGER. The order of the matrix *T* ($n \geq 0$).

d, *e*, *work*
REAL for sstevd
DOUBLE PRECISION for dstevd.
Arrays:
d(*) contains the *n* diagonal elements of the tridiagonal matrix *T*.
The dimension of *d* must be at least max(1, *n*).
e(*) contains the *n*-1 off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*). The *n*-th element of this array is used as workspace.
work(*) is a workspace array.

The dimension of *work* must be at least *lwork*.

ldz INTEGER. The leading dimension of the output array *z*. Constraints:

ldz≥ 1 if *job* = 'N';

ldz≥ max(1, *n*) if *job* = 'V'.

lwork INTEGER.

The dimension of the array *work*.

Constraints:

if *jobz* = 'N' or $n \leq 1$, then *lwork* ≥ 1;

if *jobz* = 'V' and $n > 1$, then *lwork* ≥$n^2 + 4*n + 1$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.

The dimension of the array *iwork*.

Constraints:

if *jobz* = 'N' or $n \leq 1$, then *liwork* ≥ 1;

if *jobz* = 'V' and $n > 1$, then *liwork* ≥ 5*n+3.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

d On exit, if *info* = 0, contains the eigenvalues of the matrix *T* in ascending order.

See also *info*.

z REAL for sstevd

DOUBLE PRECISION for dstevd

Array, size (*ldz*,*) .

The second dimension of *z* must be:

at least 1 if *jobz* = 'N';

at least max(1, *n*) if *jobz* = 'V'.

If *jobz* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *T*.

If *jobz* = 'N', then *z* is not referenced.

e	On exit, this array is overwritten with intermediate results.
work(1)	On exit, if *lwork* > 0, then *work*(1) returns the required minimal size of *lwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine stevd interface are the following:

d	Holds the vector of length *n*.
e	Holds the vector of length *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $||E||_2 = O(\varepsilon)*||T||_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$|\mu_i - \lambda_i| \leq c(n)*\varepsilon*||T||_2$

where $c(n)$ is a modestly increasing function of *n*.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$\theta(z_i, w_i) \leq c(n)*\varepsilon*||T||_2 / \min_{i \neq j}|\lambda_i - \lambda_j|$.

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stevx

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call dstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call stevx(d, e, w [, z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *job* = 'N', then only eigenvalues are computed.

If *job* = 'V', then eigenvalues and eigenvectors are computed.

range

CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *w(i)* in the half-open interval: $vl < w(i) \le vu$.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

n

INTEGER. The order of the matrix *A* ($n \ge 0$).

d, *e*, *work*

REAL for sstevx

DOUBLE PRECISION for dstevx.

Arrays:

d(*) contains the *n* diagonal elements of the tridiagonal matrix *A*.

The dimension of *d* must be at least max(1, *n*).

e(*) contains the *n*-1 subdiagonal elements of *A*.

The dimension of *e* must be at least max(1, *n*-1). The *n*-th element of this array is used as workspace.

work(*) is a workspace array.

The dimension of *work* must be at least max(1, 5*n*).

vl, *vu*

REAL for sstevx

DOUBLE PRECISION for dstevx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl*< *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for sstevx

DOUBLE PRECISION for dstevx. The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldz

INTEGER. The leading dimensions of the output array *z*; *ldz*≥ 1. If *jobz* = 'V', then *ldz*≥ max(1, *n*).

iwork

INTEGER. Workspace array, size at least max(1, 5*n*).

Output Parameters

m

INTEGER. The total number of eigenvalues found,

0 ≤*m*≤*n*.

If *range* = 'A', *m* = *n*, if *range* = 'I', *m* = *iu*-*il*+1, and if *range* = 'V' the exact value of *m* is unknown.

w, *z*

REAL for sstevx

DOUBLE PRECISION for dstevx.

Arrays:

w(*), size at least max(1, *n*).

The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.

z(*ldz*,*) .

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

d, *e*

On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

ifail

INTEGER.

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine stevx interface are the following:

d

Holds the vector of length *n*.

e

Holds the vector of length *n*.

w

Holds the vector of length *n*.

z

Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.

ifail

Holds the vector of length *n*.

vl

Default value for this element is *vl* = -HUGE(*vl*).

vu

Default value for this element is *vu* = HUGE(*vl*).

il

Default value for this argument is *il* = 1.

iu

Default value for this argument is *iu* = *n*.

abstol

Default value for this element is *abstol* = 0.0_WP.

jobz

Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted

Note that there will be an error condition if *ifail* is present and *z* is omitted.

range

Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:

range = 'V', if one of or both *vl* and *vu* are present,

range = 'I', if one of or both *il* and *iu* are present,

range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*$|A|_1$ is used instead. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, set *abstol* to 2*?lamch('S').

?stevr
Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

Syntax

call sstevr(*jobz*, *range*, *n*, *d*, *e*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call dstevr(*jobz*, *range*, *n*, *d*, *e*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call stevr(*d*, *e*, *w* [, *z*] [,*vl*] [,*vu*] [,*il*] [,*iu*] [,*m*] [,*isuppz*] [,*abstol*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, the routine calls stemr to compute the eigenspectrum using Relatively Robust Representations. stegr computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L*D*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of *T*:

a. Compute $T - \sigma_i = L_i*D_i*L_i^T$, such that $L_i*D_i*L_i^T$ is a relatively robust representation.
b. Compute the eigenvalues, λ_j, of $L_i*D_i*L_i^T$ to high relative accuracy by the *dqds* algorithm.
c. If there is a cluster of close eigenvalues, "choose" σ_i close to the cluster, and go to Step (a).

d. Given the approximate eigenvalue λ_j of $L_i*D_i*L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?stevr` calls stemr when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. `?stevr` calls stebz and stein on non-IEEE machines and when partial spectrum requests are made.

Input Parameters

jobz

`CHARACTER*1`. Must be `'N'` or `'V'`.

If *jobz* = `'N'`, then only eigenvalues are computed.

If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

range

`CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.

If *range* = `'A'`, the routine computes all eigenvalues.

If *range* = `'V'`, the routine computes eigenvalues *w*(*i*) in the half-open interval:

vl<*w*(*i*)≤*vu*.

If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

For *range* = `'V'` or `'I'` and *iu*-*il* < *n*–1, `sstebz`/`dstebz` and `sstein`/`dstein` are called.

n

`INTEGER`. The order of the matrix *T* (*n*≥ 0).

d, *e*, *work*

`REAL` for `sstevr`

`DOUBLE PRECISION` for `dstevr`.

Arrays:

d(*) contains the *n* diagonal elements of the tridiagonal matrix *T*.

The dimension of *d* must be at least max(1, *n*).

e(*) contains the *n*-1 subdiagonal elements of *A*.

The dimension of *e* must be at least max(1, *n*-1). The *n*-th element of this array is used as workspace.

work is a workspace array, its dimension max(1, *lwork*).

vl, *vu*

`REAL` for `sstevr`

`DOUBLE PRECISION` for `dstevr`.

If *range* = `'V'`, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl*< *vu*.

If *range* = `'A'` or `'I'`, *vl* and *vu* are not referenced.

il, *iu*

`INTEGER`.

If *range* = `'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for sstevr

DOUBLE PRECISION for dstevr.

The absolute error tolerance to which each eigenvalue/eigenvector is required.

If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*. If *abstol* < *n* *eps*||*T*||, then *n* *eps*||*T*|| will be used in its place, where eps is the machine precision, and ||*T*|| is the 1-norm of the matrix *T*. The eigenvalues are computed to an accuracy of eps*||*T*|| irrespective of *abstol*.

If high relative accuracy is important, set *abstol* to ?lamch('S').

ldz

INTEGER. The leading dimension of the output array *z*.

Constraints:

ldz≥ 1 if *jobz* = 'N';

ldz≥ max(1, *n*) if *jobz* = 'V'.

lwork

INTEGER.

The dimension of the array *work*. Constraint:

lwork≥ max(1, 20**n*).

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork

INTEGER.

Workspace array, its dimension max(1, *liwork*).

liwork

INTEGER.

The dimension of the array *iwork*,

lwork≥ max(1, 10**n*).

If *liwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

m

INTEGER. The total number of eigenvalues found,

0 ≤*m*≤*n*. If *range* = 'A', *m* = *n*, if *range* = 'I', *m* = *iu*−*il*+1, and if *range* = 'V' the exact value of *m* is unknown..

w, *z*

REAL for sstevr

DOUBLE PRECISION for dstevr.

Arrays:

w(*), size at least max(1, *n*).

The first *m* elements of *w* contain the selected eigenvalues of the matrix *T* in ascending order.

z(*ldz*,*).

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

d, *e*

On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

isuppz

INTEGER.

Array, size at least 2 *max(1, *m*).

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*).

Implemented only for *range* = 'A' or 'I' and *iu*-*il* = *n*-1.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, an internal error has occurred.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine stevr interface are the following:

d

Holds the vector of length *n*.

e

Holds the vector of length *n*.

w

Holds the vector of length *n*.

z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
isuppz	Holds the vector of length (2**n*), where the values (2**m*) are significant.
vl	Default value for this element is *vl* = –HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

Normal execution of the routine ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = –1 (*liwork* = –1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = –1 (*liwork* = –1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Nonsymmetric Eigenvalue Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving nonsymmetric eigenproblems. See also computational routines that can be called to solve these problems.

Table "Driver Routines for Solving Nonsymmetric Eigenproblems" lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Nonsymmetric Eigenproblems

Routine Name	Operation performed
gees	Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.
geesx	Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.
geev	Computes the eigenvalues and left and right eigenvectors of a general matrix.
geevx	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

?gees
Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.

Syntax

```
call sgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work, lwork, bwork, info)
call dgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work, lwork, bwork, info)
call cgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork, rwork, bwork, info)
call zgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork, rwork, bwork, info)
call gees(a, wr, wi [,vs] [,select] [,sdim] [,info])
call gees(a, w [,vs] [,select] [,sdim] [,info])
```

Include Files

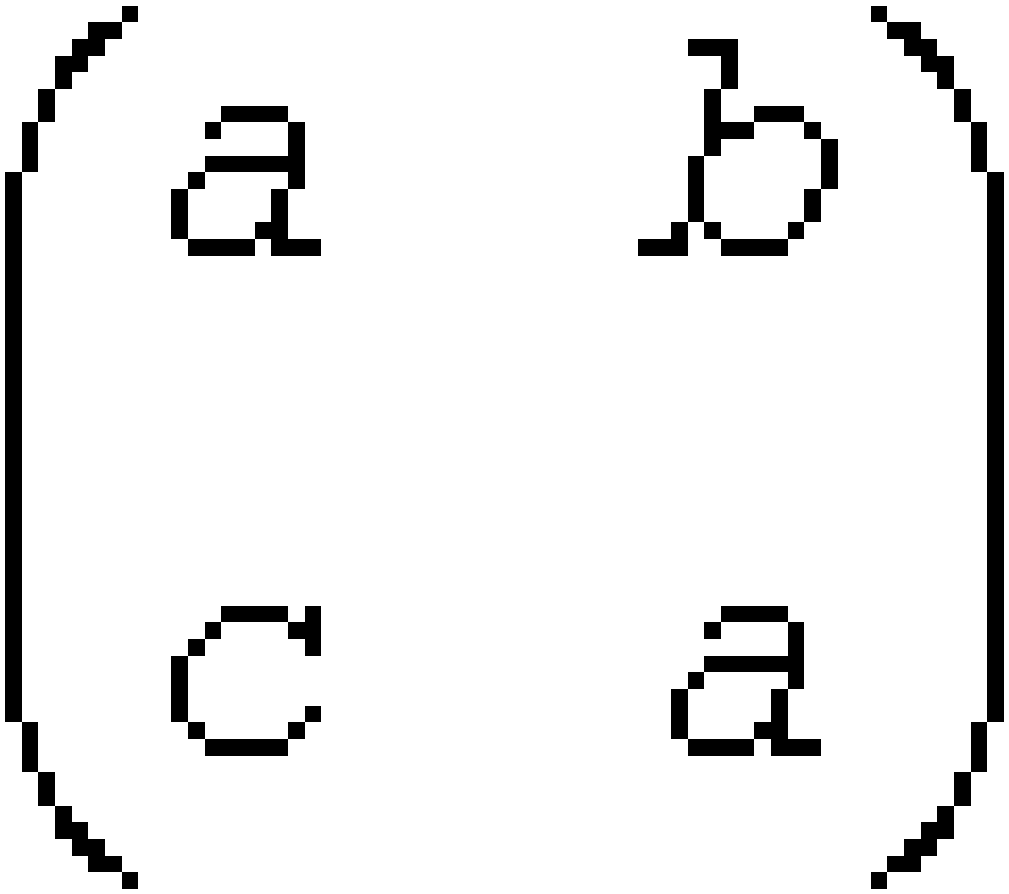
- mkl.fi, lapack.f90

Description

The routine computes for an *n*-by-*n* real/complex nonsymmetric matrix *A*, the eigenvalues, the real Schur form *T*, and, optionally, the matrix of Schur vectors *Z*. This gives the Schur factorization $A = Z*T*Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left. The leading columns of *Z* then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form



where *b*c* < 0. The eigenvalues of such a block are

$$a \pm i\sqrt{bc}$$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

jobvs

CHARACTER*1. Must be 'N' or 'V'.

If *jobvs* = 'N', then Schur vectors are not computed.

If *jobvs* = 'V', then Schur vectors are computed.

sort

CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.

If *sort* = 'N', then eigenvalues are not ordered.

If *sort* = 'S', eigenvalues are ordered (see *select*).

select

LOGICAL FUNCTION of two REAL arguments for real flavors.

LOGICAL FUNCTION of one COMPLEX argument for complex flavors.

select must be declared EXTERNAL in the calling subroutine.

If *sort* = 'S', *select* is used to select eigenvalues to sort to the top left of the Schur form.

If *sort* = 'N', *select* is not referenced.

For real flavors:

An eigenvalue *wr*(j)+sqrt(-1)**wi*(j) is selected if *select*(*wr*(j), *wi*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

For complex flavors:

An eigenvalue *w*(j) is selected if *select*(*w*(j) is true.

Note that a selected complex eigenvalue may no longer satisfy *select*(*wr*(j), *wi*(j))= .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* may be set to *n*+2 (see *info* below).

n

INTEGER. The order of the matrix *A* (*n*≥ 0).

a, *work*

REAL for sgees

DOUBLE PRECISION for dgees

COMPLEX for cgees

DOUBLE COMPLEX for zgees.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).

ldvs

INTEGER. The leading dimension of the output array *vs*. Constraints:

ldvs≥ 1;

ldvs≥ max(1, *n*) if *jobvs* = 'V'.

lwork

INTEGER.

The dimension of the array *work*.

Constraint:

lwork≥ max(1, 3*n*) for real flavors;

lwork≥ max(1, 2*n*) for complex flavors.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork

REAL for cgees

DOUBLE PRECISION for zgees

Workspace array, size at least max(1, *n*). Used in complex flavors only.

bwork LOGICAL. Workspace array, size at least max(1, *n*). Not referenced if *sort* = 'N'.

Output Parameters

a On exit, this array is overwritten by the real-Schur/Schur form *T*.

sdim INTEGER.

If *sort* = 'N', *sdim*= 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.

Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, *wi* REAL for sgees

DOUBLE PRECISION for dgees

Arrays, size at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w COMPLEX for cgees

DOUBLE COMPLEX for zgees.

Array, size at least max(1, *n*). Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.

vs REAL for sgees

DOUBLE PRECISION for dgees

COMPLEX for cgees

DOUBLE COMPLEX for zgees.

Array *vs*(*ldvs*,*);the second dimension of *vs* must be at least max(1, *n*).

If *jobvs* = 'V', *vs* contains the orthogonal/unitary matrix *Z* of Schur vectors.

If *jobvs* = 'N', *vs* is not referenced.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, and

i≤*n*:

the *QR* algorithm failed to compute all the eigenvalues; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged; if `jobvs = 'V'`, *vs* contains the matrix which reduces *A* to its partially converged Schur form;

`i = n+1`:

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

i = *n*+2:

after reordering, round-off changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy `select = .TRUE.`. This could also be caused by underflow due to scaling.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gees` interface are the following:

`a`	Holds the matrix *A* of size (*n*, *n*).
`wr`	Holds the vector of length *n*. Used in real flavors only.
`wi`	Holds the vector of length *n*. Used in real flavors only.
`w`	Holds the vector of length *n*. Used in complex flavors only.
`vs`	Holds the matrix *VS* of size (*n*, *n*).
`jobvs`	Restored based on the presence of the argument *vs* as follows: `jobvs = 'V'`, if *vs* is present, `jobvs = 'N'`, if *vs* is omitted.
`sort`	Restored based on the presence of the argument *select* as follows: `sort = 'S'`, if *select* is present, `sort = 'N'`, if *select* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geesx
Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

Syntax

`call sgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs, rconde, rcondv, work, lwork, iwork, liwork, bwork, info)`

`call dgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs, rconde, rcondv, work, lwork, iwork, liwork, bwork, info)`

`call cgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde, rcondv, work, lwork, rwork, bwork, info)`

`call zgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde, rcondv, work, lwork, rwork, bwork, info)`

`call geesx(a, wr, wi [,vs] [,select] [,sdim] [,rconde] [,rcondev] [,info])`

`call geesx(a, w [,vs] [,select] [,sdim] [,rconde] [,rcondev] [,info])`

Include Files

- `mkl.fi, lapack.f90`

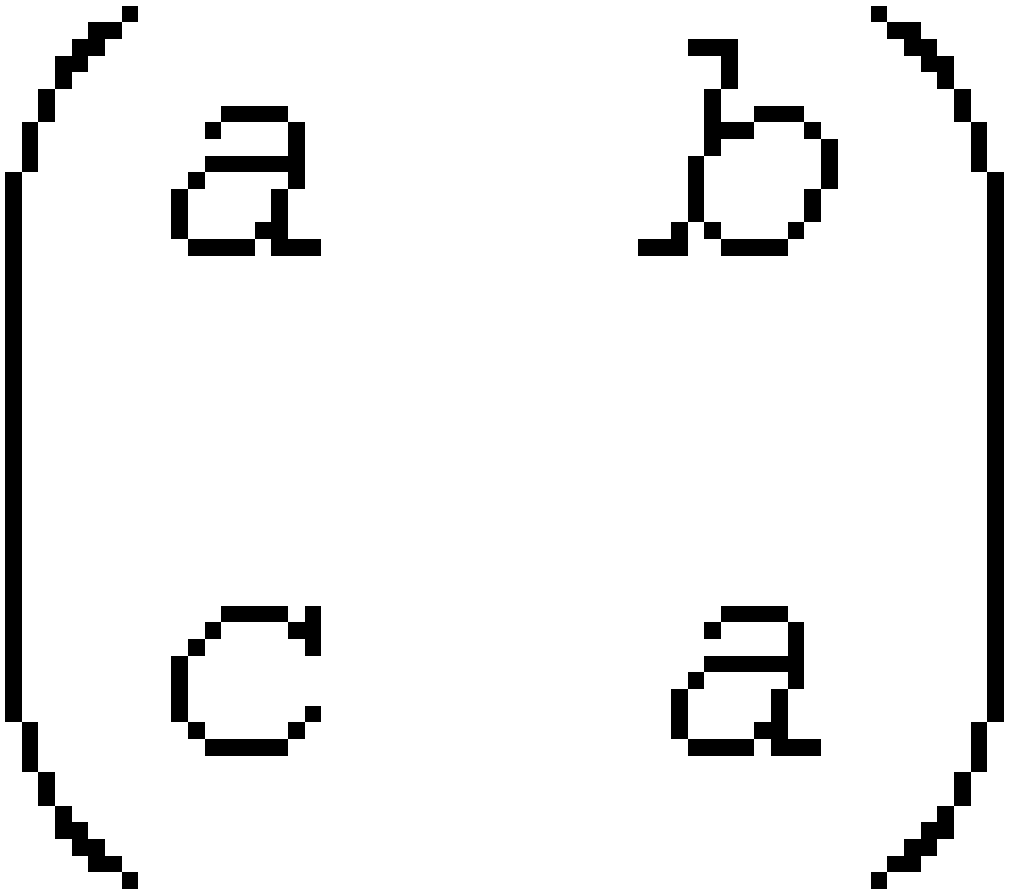
Description

The routine computes for an *n*-by-*n* real/complex nonsymmetric matrix *A*, the eigenvalues, the real-Schur/Schur form *T*, and, optionally, the matrix of Schur vectors *Z*. This gives the Schur factorization $A = Z*T*Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (*rcondv*). The leading columns of *Z* form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers *rconde* and *rcondv*, see [LUG], Section 4.10 (where these quantities are called *s* and *sep* respectively).

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form



where *b*c* < 0. The eigenvalues of such a block are

$$a \pm i\sqrt{bc}$$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

jobvs

CHARACTER*1. Must be 'N' or 'V'.

If *jobvs* = 'N', then Schur vectors are not computed.

If *jobvs* = 'V', then Schur vectors are computed.

sort

CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.

If *sort* = 'N', then eigenvalues are not ordered.

If *sort* = 'S', eigenvalues are ordered (see *select*).

select

LOGICAL FUNCTION of two REAL arguments for real flavors.

LOGICAL FUNCTION of one COMPLEX argument for complex flavors.

select must be declared EXTERNAL in the calling subroutine.

If *sort* = 'S', *select* is used to select eigenvalues to sort to the top left of the Schur form.

If *sort* = 'N', *select* is not referenced.

For real flavors:

An eigenvalue *wr*(j)+sqrt(-1)**wi*(j) is selected if *select*(*wr*(j), *wi*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

For complex flavors:

An eigenvalue *w*(j) is selected if *select*(*w*(j)) is true.

Note that a selected complex eigenvalue may no longer satisfy *select*(*wr*(j), *wi*(j))= .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* may be set to *n*+2 (see *info* below).

sense — CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If *sense* = 'N', none are computed;

If *sense* = 'E', computed for average of selected eigenvalues only;

If *sense* = 'V', computed for selected right invariant subspace only;

If *sense* = 'B', computed for both.

If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.

n — INTEGER. The order of the matrix *A* (*n*≥ 0).

a, *work* — REAL for sgeesx

DOUBLE PRECISION for dgeesx

COMPLEX for cgeesx

DOUBLE COMPLEX for zgeesx.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).

ldvs — INTEGER. The leading dimension of the output array *vs*. Constraints:

ldvs≥ 1;

ldvs≥ max(1, *n*) if *jobvs* = 'V'.

lwork — INTEGER.

The dimension of the array *work*. Constraint:

lwork≥ max(1, 3*n*) for real flavors;

lwork≥ max(1, 2*n*) for complex flavors.

Also, if *sense* = 'E', 'V', or 'B', then

lwork≥*n*+2**sdim***(*n*-*sdim*) for real flavors;

lwork≥ 2**sdim***(*n*-*sdim*) for complex flavors;

where *sdim* is the number of selected eigenvalues computed by this routine.

Note that 2**sdim***(*n*-*sdim*) ≤*n***n*/2. Note also that an error is only returned if *lwork*<max(1, 2**n*), but if *sense* = 'E', or 'V', or 'B' this may not be large enough.

For good performance, *lwork* must generally be larger.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates upper bound on the optimal size of the array *work*, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

iwork INTEGER.

Workspace array, size (*liwork*). Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.

liwork INTEGER.

The dimension of the array *iwork*. Used in real flavors only.

Constraint:

liwork≥ 1;

if *sense* = 'V' or 'B', *liwork*≥*sdim***(*n*-*sdim*).

rwork REAL for cgeesx

DOUBLE PRECISION for zgeesx

Workspace array, size at least max(1, *n*). Used in complex flavors only.

bwork LOGICAL. Workspace array, size at least max(1, *n*). Not referenced if *sort* = 'N'.

Output Parameters

a On exit, this array is overwritten by the real-Schur/Schur form *T*.

sdim INTEGER.

If *sort* = 'N', *sdim*= 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.

Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, *wi* REAL for sgeesx

DOUBLE PRECISION for dgeesx

Arrays, size at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for cgeesx

DOUBLE COMPLEX for zgeesx.

Array, size at least max(1, *n*). Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.

vs

REAL for sgeesx

DOUBLE PRECISION for dgeesx

COMPLEX for cgeesx

DOUBLE COMPLEX for zgeesx.

Array *vs*(*ldvs*,*); the second dimension of *vs* must be at least max(1, *n*).

If *jobvs* = 'V', *vs* contains the orthogonal/unitary matrix *Z* of Schur vectors.

If *jobvs* = 'N', *vs* is not referenced.

rconde, *rcondv*

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

If *sense* = 'E' or 'B', *rconde* contains the reciprocal condition number for the average of the selected eigenvalues.

If *sense* = 'N' or 'V', *rconde* is not referenced.

If *sense* = 'V' or 'B', *rcondv* contains the reciprocal condition number for the selected right invariant subspace.

If *sense* = 'N' or 'E', *rcondv* is not referenced.

work(1)

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*th parameter had an illegal value.

If *info* = *i*, and

i≤*n*:

the *QR* algorithm failed to compute all the eigenvalues; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged; if *jobvs* = 'V', *vs* contains the transformation which reduces *A* to its partially converged Schur form;

i = *n*+1:

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

i = *n*+2:

after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy *select* = .TRUE.. This could also be caused by underflow due to scaling.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `geesx` interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
wr	Holds the vector of length (*n*). Used in real flavors only.
wi	Holds the vector of length (*n*). Used in real flavors only.
w	Holds the vector of length (*n*). Used in complex flavors only.
vs	Holds the matrix *VS* of size (*n*, *n*).
jobvs	Restored based on the presence of the argument *vs* as follows: *jobvs* = 'V', if *vs* is present, *jobvs* = 'N', if *vs* is omitted.
sort	Restored based on the presence of the argument *select* as follows: *sort* = 'S', if *select* is present, *sort* = 'N', if *select* is omitted.
sense	Restored based on the presence of arguments *rconde* and `rcondv` as follows: *sense* = 'B', if both *rconde* and *rcondv* are present, *sense* = 'E', if *rconde* is present and *rcondv* omitted, *sense* = 'V', if *rconde* is omitted and *rcondv* present, *sense* = 'N', if both *rconde* and *rcondv* are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geev
Computes the eigenvalues and left and right eigenvectors of a general matrix.

Syntax

```
call sgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info)
call dgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info)
```

```
call cgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
call zgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
call geev(a, wr, wi [,vl] [,vr] [,info])
call geev(a, w [,vl] [,vr] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes for an n-by-n real/complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors. The right eigenvector v of A satisfies

$A*v = \lambda*v$

where λ is its eigenvalue.

The left eigenvector u of A satisfies

$u^H*A = \lambda*u^H$

where u^H denotes the conjugate transpose of u. The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Input Parameters

jobvl	CHARACTER*1. Must be 'N' or 'V'. If *jobvl* = 'N', then left eigenvectors of *A* are not computed. If *jobvl* = 'V', then left eigenvectors of *A* are computed.
jobvr	CHARACTER*1. Must be 'N' or 'V'. If *jobvr* = 'N', then right eigenvectors of *A* are not computed. If *jobvr* = 'V', then right eigenvectors of *A* are computed.
n	INTEGER. The order of the matrix *A* ($n \geq 0$).
a, *work*	REAL for sgeev DOUBLE PRECISION for dgeev COMPLEX for cgeev DOUBLE COMPLEX for zgeev. Arrays: *a*(*lda*,*) is an array containing the n-by-n matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).
ldvl, *ldvr*	INTEGER. The leading dimensions of the output arrays *vl* and *vr*, respectively.

Constraints:

ldvl≥ 1; *ldvr*≥ 1.

If *jobvl* = 'V', *ldvl*≥ max(1, *n*);

If *jobvr* = 'V', *ldvr*≥ max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*.

Constraint for real flavors:	*lwork*≥ max(1, 3*n*). If computing eigenvectors (*jobvl* = 'V' or *jobvr* = 'V'), *lwork*≥ max(1, 4*n*).
Constraint for complex flavors:	*lwork*≥ max(1, 2*n*).

For good performance, *lwork* must generally be larger.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork

REAL for cgeev

DOUBLE PRECISION for zgeev

Workspace array, size at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

a

On exit, this array is overwritten.

wr, *wi*

REAL for sgeev

DOUBLE PRECISION for dgeev

Arrays, size at least max (1, *n*) each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for cgeev

DOUBLE COMPLEX for zgeev.

Array, size at least max(1, *n*).

Contains the computed eigenvalues.

vl, *vr*

REAL for sgeev

DOUBLE PRECISION for dgeev

COMPLEX for cgeev

DOUBLE COMPLEX for zgeev.

Arrays:

vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*).

If *jobvl* = 'N', *vl* is not referenced.

For real flavors:

If the *j*-th eigenvalue is real, then u_j = *vl*(:,*j*), the *j*-th column of *vl*.

If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for i = sqrt(-1), u_j = *vl*(:,*j*) + i**vl*(:,*j*+1) and u_{j+1} = *vl*(:,*j*)- i**vl*(:,*j*+1).

For complex flavors:

u_j = *vl*(:,*j*), the *j*-th column of *vl*.

vr(*ldvr*,*); the second dimension of *vr* must be at least max(1, *n*).

If *jobvr* = 'N', *vr* is not referenced.

For real flavors:

If the *j*-th eigenvalue is real, then v_j = *vr*(:,*j*), the *j*-th column of *vr*.

If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for *i* = sqrt(-1), v_j = *vr*(:,*j*) + i**vr*(:,*j*+1) and v_{j+1} = *vr*(:,*j*) - i**vr*(:,*j*+1).

For complex flavors:

v_j = *vr*(:,*j*), the *j*-th column of *vr*.

work(1) On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; elements *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine geev interface are the following:

a Holds the matrix *A* of size (*n*, *n*).

wr Holds the vector of length *n*. Used in real flavors only.

wi Holds the vector of length *n*. Used in real flavors only.

w Holds the vector of length *n*. Used in complex flavors only.

vl Holds the matrix *VL* of size (*n*, *n*).

vr Holds the matrix *VR* of size (*n*, *n*).

jobvl	Restored based on the presence of the argument *vl* as follows: *jobvl* = 'V', if *vl* is present, *jobvl* = 'N', if *vl* is omitted.
jobvr	Restored based on the presence of the argument *vr* as follows: *jobvr* = 'V', if *vr* is present, *jobvr* = 'N', if *vr* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine exits immediately with an error and does not provide any information on the recommended workspace.

?geevx

Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

Syntax

call sgeevx(*balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *wr*, *wi*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *scale*, *abnrm*, *rconde*, *rcondv*, *work*, *lwork*, *iwork*, *info*)

call dgeevx(*balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *wr*, *wi*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *scale*, *abnrm*, *rconde*, *rcondv*, *work*, *lwork*, *iwork*, *info*)

call cgeevx(*balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *w*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *scale*, *abnrm*, *rconde*, *rcondv*, *work*, *lwork*, *rwork*, *info*)

call zgeevx(*balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *w*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *scale*, *abnrm*, *rconde*, *rcondv*, *work*, *lwork*, *rwork*, *info*)

call geevx(*a*, *wr*, *wi* [,*vl*] [,*vr*] [,*balanc*] [,*ilo*] [,*ihi*] [,*scale*] [,*abnrm*] [, *rconde*] [,*rcondv*] [,*info*])

call geevx(*a*, *w* [,*vl*] [,*vr*] [,*balanc*] [,*ilo*] [,*ihi*] [,*scale*] [,*abnrm*] [,*rconde*] [, *rcondv*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes for an *n*-by-*n* real/complex nonsymmetric matrix *A*, the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *scale*, and *abnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

The right eigenvector v of *A* satisfies

$A \cdot v = \lambda \cdot v$

where λ is its eigenvalue.

The left eigenvector u of *A* satisfies

$u^H A = \lambda u^H$

where u^H denotes the conjugate transpose of u. The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation *D***A**inv(*D*), where *D* is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see [LUG], Section 4.10.

Input Parameters

balanc

CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues.

If *balanc* = 'N', do not diagonally scale or permute;

If *balanc* = 'P', perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale;

If *balanc* = 'S', diagonally scale the matrix, i.e. replace *A* by *D***A**inv(*D*), where *D* is a diagonal matrix chosen to make the rows and columns of *A* more equal in norm. Do not permute;

If *balanc* = 'B', both diagonally scale and permute *A*.

Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

jobvl

CHARACTER*1. Must be 'N' or 'V'.

If *jobvl* = 'N', left eigenvectors of *A* are not computed;

If *jobvl* = 'V', left eigenvectors of *A* are computed.

If *sense* = 'E' or 'B', then *jobvl* must be 'V'.

jobvr

CHARACTER*1. Must be 'N' or 'V'.

If *jobvr* = 'N', right eigenvectors of *A* are not computed;

If *jobvr* = 'V', right eigenvectors of *A* are computed.

If *sense* = 'E' or 'B', then *jobvr* must be 'V'.

sense — CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If *sense* = 'N', none are computed;

If *sense* = 'E', computed for eigenvalues only;

If *sense* = 'V', computed for right eigenvectors only;

If *sense* = 'B', computed for eigenvalues and right eigenvectors.

If *sense* is 'E' or 'B', both left and right eigenvectors must also be computed (*jobvl* = 'V' and *jobvr* = 'V').

n — INTEGER. The order of the matrix *A* ($n \geq 0$).

a, *work* — REAL for sgeevx

DOUBLE PRECISION for dgeevx

COMPLEX for cgeevx

DOUBLE COMPLEX for zgeevx.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).

ldvl, *ldvr* — INTEGER. The leading dimensions of the output arrays *vl* and *vr*, respectively.

Constraints:

ldvl ≥ 1; *ldvr* ≥ 1.

If *jobvl* = 'V', *ldvl* $\geq$ max(1, *n*);

If *jobvr* = 'V', *ldvr* $\geq$ max(1, *n*).

lwork — INTEGER.

The dimension of the array *work*.

For real flavors:

If *sense* = 'N' or 'E', *lwork* $\geq$ max(1, $2n$), and if *jobvl* = 'V' or *jobvr* = 'V', *lwork* $\geq 3n$;

If *sense* = 'V' or 'B', *lwork* $\geq n*(n+6)$.

For good performance, *lwork* must generally be larger.

For complex flavors:

If *sense* = 'N'or 'E', *lwork* $\geq$ max(1, $2n$);

If *sense* = 'V' or 'B', *lwork* $\geq n^2+2n$. For good performance, *lwork* must generally be larger.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork

REAL for cgeevx

DOUBLE PRECISION for zgeevx

Workspace array, size at least max(1, 2*n*). Used in complex flavors only.

iwork

INTEGER.

Workspace array, size at least max(1, 2*n*-2). Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.

Output Parameters

a

On exit, this array is overwritten.

If *jobvl* = 'V' or *jobvr* = 'V', it contains the real-Schur/Schur form of the balanced version of the input matrix *A*.

wr, *wi*

REAL for sgeevx

DOUBLE PRECISION for dgeevx

Arrays, size at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for cgeevx

DOUBLE COMPLEX for zgeevx.

Array, size at least max(1, *n*). Contains the computed eigenvalues.

vl, *vr*

REAL for sgeevx

DOUBLE PRECISION for dgeevx

COMPLEX for cgeevx

DOUBLE COMPLEX for zgeevx.

Arrays:

vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*).

If *jobvl* = 'N', *vl* is not referenced.

For real flavors:

If the *j*-th eigenvalue is real, then u_j = *vl*(:,*j*), the *j*-th column of *vl*.

If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for i = sqrt(-1), u_j = *vl*(:,*j*) + *i***vl*(:,*j*+1) and u_{j+1} = *vl*(:,*j*)- *i***vl*(:,*j*+1).

For complex flavors:

u_j = *vl*(:,*j*), the *j*-th column of *vl*.

vr(*ldvr*,*); the second dimension of *vr* must be at least max(1, *n*).

If *jobvr* = 'N', *vr* is not referenced.

For real flavors:

If the *j*-th eigenvalue is real, then v_j = *vr*(:,*j*), the *j*-th column of *vr*.

If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for *i* = sqrt(-1), v_j = *vr*(:,*j*) + *i***vr*(:,*j*+1) and v_{j+1} = *vr*(:,*j*) - *i***vr*(:,*j*+1).

For complex flavors:

v_j = *vr*(:,*j*), the *j*-th column of *vr*.

ilo, *ihi*

INTEGER. *ilo* and *ihi* are integer values determined when *A* was balanced.

The balanced *A*(i,j) = 0 if i > j and j = 1,..., *ilo*-1 or i = *ihi*+1,..., *n*.

If *balanc* = 'N' or 'S', *ilo* = 1 and *ihi* = *n*.

scale

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size at least max(1, *n*). Details of the permutations and scaling factors applied when balancing *A*.

If *P*(*j*) is the index of the row and column interchanged with row and column *j*, and *D*(*j*) is the scaling factor applied to row and column *j*, then

scale(*j*) = *P*(*j*), for *j* = 1,...,*ilo*-1

= *D*(*j*), for *j* = *ilo*,...,*ihi*

= *P*(*j*) for *j* = *ihi*+1,..., *n*.

The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

abnrm

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

rconde, *rcondv*

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Arrays, size at least max(1, *n*) each.

rconde(*j*) is the reciprocal condition number of the *j*-th eigenvalue.

rcondv(*j*) is the reciprocal condition number of the *j*-th right eigenvector.

work(1)

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, the *QR* algorithm failed to compute all the eigenvalues, and no eigenvectors or condition numbers have been computed; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain eigenvalues which have converged.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `geevx` interface are the following:

a Holds the matrix *A* of size (*n*, *n*).

wr Holds the vector of length *n*. Used in real flavors only.

wi Holds the vector of length *n*. Used in real flavors only.

w Holds the vector of length *n*. Used in complex flavors only.

vl Holds the matrix *VL* of size (*n*, *n*).

vr Holds the matrix *VR* of size (*n*, *n*).

scale Holds the vector of length *n*.

rconde Holds the vector of length *n*.

rcondv Holds the vector of length *n*.

balanc Must be `'N'`, `'B'`, `'P'` or `'S'`. The default value is `'N'`.

jobvl Restored based on the presence of the argument *vl* as follows:

jobvl = `'V'`, if *vl* is present,

jobvl = `'N'`, if *vl* is omitted.

jobvr Restored based on the presence of the argument *vr* as follows:

jobvr = `'V'`, if *vr* is present,

jobvr = `'N'`, if *vr* is omitted.

sense Restored based on the presence of arguments *rconde* and `rcondv` as follows:

sense = `'B'`, if both *rconde* and *rcondv* are present,

sense = `'E'`, if *rconde* is present and *rcondv* omitted,

sense = `'V'`, if *rconde* is omitted and *rcondv* present,

sense = `'N'`, if both *rconde* and *rcondv* are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork = -1*, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Singular Value Decomposition: LAPACK Driver Routines

Table "Driver Routines for Singular Value Decomposition" lists the LAPACK driver routines that perform singular value decomposition for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are the same except that the first character is removed.

Driver Routines for Singular Value Decomposition

Routine Name	Operation performed
?gesvd	Computes the singular value decomposition of a general rectangular matrix.
?gesdd	Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.
?gejsv	Computes the singular value decomposition of a real matrix using a preconditioned Jacobi SVD method.
?gesvj	Computes the singular value decomposition of a real matrix using Jacobi plane rotations.
?ggsvd	Computes the generalized singular value decomposition of a pair of general rectangular matrices.
?gesvdx	Computes the SVD and left and right singular vectors for a matrix.
?bdsvdx	Computes the SVD of a bidiagonal matrix.

Singular Value Decomposition - LAPACK Computational Routines

?gesvd
Computes the singular value decomposition of a general rectangular matrix.

Syntax

```
call sgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
call dgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
call cgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
call zgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
call gesvd(a, s [,u] [,vt] [,ww] [,job] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the singular value decomposition (SVD) of a real/complex *m*-by-*n* matrix *A*, optionally computing the left and/or right singular vectors. The SVD is written as

$A = U*\Sigma*V^T$ for real routines

$A = U*\Sigma*V^H$ for complex routines

where Σ is an *m*-by-*n* matrix which is zero except for its min(*m*,*n*) diagonal elements, *U* is an *m*-by-*m* orthogonal/unitary matrix, and *V* is an *n*-by-*n* orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of *A*; they are real and non-negative, and are returned in descending order. The first min(*m*, *n*) columns of *U* and *V* are the left and right singular vectors of *A*.

The routine returns V^T (for real flavors) or V^H (for complex flavors), not *V*.

Input Parameters

jobu

`CHARACTER*1`. Must be `'A'`, `'S'`, `'O'`, or `'N'`. Specifies options for computing all or part of the matrix *U*.

If *jobu* = `'A'`, all *m* columns of *U* are returned in the array *u*;

if *jobu* = `'S'`, the first min(*m*, *n*) columns of *U* (the left singular vectors) are returned in the array *u*;

if *jobu* = `'O'`, the first min(*m*, *n*) columns of *U* (the left singular vectors) are overwritten on the array *a*;

if *jobu* = `'N'`, no columns of *U* (no left singular vectors) are computed.

jobvt

`CHARACTER*1`. Must be `'A'`, `'S'`, `'O'`, or `'N'`. Specifies options for computing all or part of the matrix V^T/V^H.

If *jobvt* = `'A'`, all *n* rows of V^T/V^H are returned in the array *vt*;

if *jobvt* = `'S'`, the first min(*m*,*n*) rows of V^T/V^H (the right singular vectors) are returned in the array *vt*;

if *jobvt* = `'O'`, the first min(*m*,*n*) rows of V^T/V^H) (the right singular vectors) are overwritten on the array *a*;

if *jobvt* = `'N'`, no rows of V^T/V^H (no right singular vectors) are computed.

jobvt and *jobu* cannot both be `'O'`.

m

`INTEGER`. The number of rows of the matrix *A* (*m*≥ 0).

n

`INTEGER`. The number of columns in *A* (*n*≥ 0).

a, *work*

`REAL` for `sgesvd`

`DOUBLE PRECISION` for `dgesvd`

`COMPLEX` for `cgesvd`

`DOUBLE COMPLEX` for `zgesvd`.

Arrays:

a(*lda*,*) is an array containing the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

lda

`INTEGER`. The leading dimension of the array *a*.

Must be at least max(1, *m*) .

ldu, *ldvt*

`INTEGER`. The leading dimensions of the output arrays *u* and *vt*, respectively.

Constraints:

ldu≥ 1; *ldvt*≥ 1.

If *jobu* = 'A'or 'S', *ldu*≥*m*;

If *jobvt* = 'A', *ldvt*≥*n*;

If *jobvt* = 'S', *ldvt*≥ min(*m*, *n*) .

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

lwork≥ 1

lwork≥ max(3*min(*m*, *n*)+max(*m*, *n*), 5*min(*m*,*n*)) (for real flavors);

lwork≥ 2*min(*m*, *n*)+max(*m*, *n*) (for complex flavors).

For good performance, *lwork* must generally be larger.

If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See *Application Notes* for details.

rwork

REAL for cgesvd

DOUBLE PRECISION for zgesvd

Workspace array, size at least max(1, 5*min(*m*, *n*)). Used in complex flavors only.

Output Parameters

a

On exit,

If *jobu* = 'O', *a* is overwritten with the first min(*m*,*n*) columns of *U* (the left singular vectors stored columnwise);

If *jobvt* = 'O', *a* is overwritten with the first min(*m*, *n*) rows of V^T/V^H (the right singular vectors stored rowwise);

If *jobu*≠'O' and *jobvt*≠'O', the contents of *a* are destroyed.

s

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, min(*m*,*n*)). Contains the singular values of *A* sorted so that *s*(*i*) ≥ *s*(*i*+1).

u, *vt*

REAL for sgesvd

DOUBLE PRECISION for dgesvd

COMPLEX for cgesvd

DOUBLE COMPLEX for zgesvd.

Arrays:

u(*ldu*,*); the second dimension of *u* must be at least max(1, *m*) if *jobu* = 'A', and at least max(1, min(*m*, *n*)) if *jobu* = 'S'.

If *jobu* = 'A', *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*.

If *jobu* = 'S', *u* contains the first min(*m*, *n*) columns of *U* (the left singular vectors stored column-wise).

If *jobu* = 'N' or 'O', *u* is not referenced.

vt(*ldvt*,*); the second dimension of *vt* must be at least max(1, *n*).

If *jobvt* = 'A', *vt* contains the *n*-by-*n* orthogonal/unitary matrix V^T/V^H.

If *jobvt* = 'S', *vt* contains the first min(*m*, *n*) rows of V^T/V^H (the right singular vectors stored row-wise).

If *jobvt* = 'N'or 'O', *vt* is not referenced.

work — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

For real flavors:

If *info* > 0, *work*(2:min(*m*,*n*)) contains the unconverged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not necessarily sorted). *B* satisfies *A*=*u***B***vt*, so it has the same singular values as *A*, and singular vectors related by *u* and *vt*.

rwork — On exit (for complex flavors), if *info* > 0, *rwork*(1:min(*m*,*n*)-1) contains the unconverged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not necessarily sorted). *B* satisfies *A* = *u***B***vt*, so it has the same singular values as *A*, and singular vectors related by *u* and *vt*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then if ?bdsqr did not converge, *i* specifies how many superdiagonals of the intermediate bidiagonal form *B* did not converge to zero (see the description of the *work* and *rwork* parameters for details).

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gesvd interface are the following:

a — Holds the matrix *A* of size (*m*, *n*).

s — Holds the vector of length min(*m*, *n*).

u — If present and is a square *m*-by-*m* matrix, on exit contains the *m*-by-*m* orthogonal/unitary matrix *U*.

Otherwise, if present, on exit contains the first min(*m*,*n*) columns of the matrix *U* (left singular vectors stored column-wise).

vt — If present and is a square *n*-by-*n* matrix, on exit contains the *n*-by-*n* orthogonal/unitary matrix V'^T/V'^H.

Otherwise, if present, on exit contains the first min(*m*,*n*) rows of the matrix V'^T/V'^H (right singular vectors stored row-wise).

ww	Holds the vector of length min(*m*, *n*)-1. *ww* contains the unconverged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not necessarily sorted). *B* satisfies *A = U*B*VT*, so it has the same singular values as *A*, and singular vectors related by *U* and *VT*.
job	Must be either 'N', or 'U', or 'V'. The default value is 'N'. If *job = 'U'*, and *u* is not present, then *u* is returned in the array *a*. If *job = 'V'*, and *vt* is not present, then *vt* is returned in the array *a*.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork = -1*.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork = -1*, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gesdd

Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.

Syntax

```
call sgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call dgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call cgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
call zgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
call gesdd(a, s [,u] [,vt] [,jobz] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the singular value decomposition (SVD) of a real/complex *m*-by-*n* matrix *A*, optionally computing the left and/or right singular vectors.

If singular vectors are desired, it uses a divide-and-conquer algorithm. The SVD is written

$A = U*\Sigma*V^{T}$ for real routines,

$A = U*\Sigma*V^{H}$ for complex routines,

where Σ is an *m*-by-*n* matrix which is zero except for its min(*m*,*n*) diagonal elements, *U* is an *m*-by-*m* orthogonal/unitary matrix, and *V* is an *n*-by-*n* orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of *A*; they are real and non-negative, and are returned in descending order. The first min(*m*, *n*) columns of *U* and *V* are the left and right singular vectors of *A*.

Note that the routine returns $vt = V^T$ (for real flavors) or $vt = V^H$ (for complex flavors), not *V*.

Input Parameters

jobz

`CHARACTER*1`. Must be `'A'`, `'S'`, `'O'`, or `'N'`.

Specifies options for computing all or part of the matrices *U* and *V*.

If *jobz* = `'A'`, all *m* columns of *U* and all *n* rows of V^T or V^H are returned in the arrays *u* and *vt*;

if *jobz* = `'S'`, the first min(*m*, *n*) columns of *U* and the first min(*m*, *n*) rows of V^T or V^H are returned in the arrays *u* and *vt*;

if *jobz* = `'O'`, then

if $m \geq n$, the first *n* columns of *U* are overwritten in the array *a* and all rows of V^T or V^H are returned in the array *vt*;

if $m < n$, all columns of *U* are returned in the array *u* and the first *m* rows of V^T or V^H are overwritten in the array *a*;

if *jobz* = `'N'`, no columns of *U* or rows of V^T or V^H are computed.

m

`INTEGER`. The number of rows of the matrix *A* ($m \geq 0$).

n

`INTEGER`. The number of columns in *A* ($n \geq 0$).

a, *work*

`REAL` for `sgesdd`

`DOUBLE PRECISION` for `dgesdd`

`COMPLEX` for `cgesdd`

`DOUBLE COMPLEX` for `zgesdd`.

Arrays:

a(*lda*,*) is an array containing the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

lda

`INTEGER`. The leading dimension of the array *a*. Must be at least max(1, *m*).

ldu, *ldvt*

`INTEGER`. The leading dimensions of the output arrays *u* and *vt*, respectively.

Constraints:

$ldu \geq 1$; $ldvt \geq 1$.

If *jobz* = `'S'` or `'A'`, or *jobz* = `'O'` and $m < n$,

then $ldu \geq m$;

If *jobz* = `'A'` or *jobz* = `'O'` and $m \geq n$,

then $ldvt \geq n$;

If *jobz* = `'S'`, $ldvt \geq \min(m, n)$.

lwork INTEGER.

The dimension of the array *work*; *lwork* ≥ 1.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the *work*(1), and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the theoretical minimum value of *lwork* when a workspace query is not performed.

rwork REAL for cgesdd

DOUBLE PRECISION for zgesdd

Workspace array, size at least max(1, 7*min(*m*,*n*)) if *jobz* = 'N'.

Otherwise, the dimension of *rwork* must be at least max(1,min(*m*,*n*)*max(5*min(*m*,*n*)+7,2*max(*m*,*n*)+2*min(*m*,*n*)+1)). This array is used in complex flavors only.

iwork INTEGER. Workspace array, size at least max(1, 8 *min(*m*, *n*)).

Output Parameters

a On exit:

If *jobz* = 'O', then if *m* ≥ *n*, *a* is overwritten with the first *n* columns of *U* (the left singular vectors, stored columnwise). If *m* < *n*, *a* is overwritten with the first *m* rows of V^T (the right singular vectors, stored rowwise);

If *jobz* ≠ 'O', the contents of *a* are destroyed.

s REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, size at least max(1, min(*m*,*n*)). Contains the singular values of *A* sorted so that *s*(i) ≥ *s*(i+1).

u, *vt* REAL for sgesdd

DOUBLE PRECISION for dgesdd

COMPLEX for cgesdd

DOUBLE COMPLEX for zgesdd.

Arrays:

u(*ldu*,*); the second dimension of *u* must be at least max(1, *m*) if *jobz* = 'A' or *jobz* = 'O' and *m* < *n*.

If *jobz* = 'S', the second dimension of *u* must be at least max(1, min(*m*, *n*)).

If *jobz* = 'A'or *jobz* = 'O' and *m* < *n*, *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*.

If *jobz* = 'S', *u* contains the first min(*m*, *n*) columns of *U* (the left singular vectors, stored columnwise).

If *jobz* = 'O' and *m* ≥ *n*, or *jobz* = 'N', *u* is not referenced.

vt(*ldvt*,*); the second dimension of *vt* must be at least max(1, *n*).

If *jobz* = 'A' or *jobz* = 'O' and *m*≥*n*, *vt* contains the *n*-by-*n* orthogonal/unitary matrix V^T.

If *jobz* = 'S', *vt* contains the first min(*m*, *n*) rows of V^T (the right singular vectors, stored rowwise).

If *jobz* = 'O' and *m* < *n*, or *jobz* = 'N', *vt* is not referenced.

work(1) — On exit, if *info* = 0, then *work(1)* returns the optimal size of *lwork*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = -4, *A* had a NAN entry.

If *info* = *i*, then ?bdsdc did not converge, updating process failed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine gesdd interface are the following:

a — Holds the matrix *A* of size (*m*, *n*).

s — Holds the vector of length min(*m*, *n*).

u — Holds the matrix *U* of size

- (*m*,*m*) if *jobz*='A' or *jobz*='O' and *m* < *n*
- (*m*,min(*m*, *n*)) if *jobz*='S'

u is not referenced if *jobz* is not supplied or if *jobz*='N' or *jobz*='O' and *m*≥*n*.

vt — Holds the matrix *VT* of size

- (*n*,*n*) if *jobz*='A' or *jobz*='O' and *m*≥*n*
- (min(*m*, *n*), *n*) if *jobz*='S'

vt is not referenced if *jobz* is not supplied or if *jobz*='N' or *jobz*='O' and *m* < *n*.

job — Must be 'N', 'A', 'S', or 'O'. The default value is 'N'.

Application Notes

The theoretical minimum value for *lwork* depends on the flavor of the routine.

For real flavors:

If *jobz* = 'N', *lwork*= 3*min(*m*, *n*) + max (max(*m*,*n*), 6*min(*m*, *n*));

If *jobz* = 'O', *lwork*= 3*(min(*m*, *n*))^2 + max (max(*m*, *n*), 5*(min(*m*, *n*))^2 + 4*min(*m*, *n*));

If *jobz* = 'S' or 'A', *lwork*= min(*m*, *n*)*(6 + 4*min(*m*, *n*)) + max(*m*, *n*);

For complex flavors:

If *jobz* = 'N', *lwork*= 2*min(*m*, *n*) + max(*m*, *n*);

If *jobz* = 'O', *lwork*= 2*(min(*m*, *n*))^2 + max(*m*, *n*) + 2*min(*m*, *n*);

If *jobz* = 'S' or 'A', *lwork*= (min(*m*, *n*))^2 + max(*m*, *n*) + 2*min(*m*, *n*);

The optimal value of *lwork* returned by a workspace query generally provides better performance than the theoretical minimum value.The value of *lwork* returned by a workspace query is generally larger than the theoretical minimum value, but for very small matrices it can be smaller. The absolute minimum value of *lwork* is the minimum of the workspace query result and the theoretical minimum.

If you set *lwork* to a value less than the absolute minimum value and not equal to -1, the routine returns immediately with an error exit and does not provide information on the recommended workspace size.

?gejsv
Computes the singular value decomposition using a preconditioned Jacobi SVD method.

Syntax

```
call sgejsv(joba, jobu, jobv, jobr, jobt, jobp, m, n, a, lda, sva, u, ldu, v, ldv, work, lwork, iwork, info)
```

```
call dgejsv(joba, jobu, jobv, jobr, jobt, jobp, m, n, a, lda, sva, u, ldu, v, ldv, work, lwork, iwork, info)
```

```
call cgejsv (joba, jobu, jobv, jobr, jobt, jobp, m, n, a, lda, sva, u, ldu, v, ldv, cwork, lwork, rwork, lrwork, iwork, info )
```

```
call zgejsv (joba, jobu, jobv, jobr, jobt, jobp, m, n, a, lda, sva, u, ldu, v, ldv, cwork, lwork, rwork, lrwork, iwork, info )
```

Include Files

- mkl.fi

Description

The routine computes the singular value decomposition (SVD) of a real/complex *m*-by-*n* matrix *A*, where *m*≥*n*.

The SVD is written as

A = *U**Σ**V*^{T}, for real routines

A = *U**Σ**V*^{H}, for complex routines

where Σ is an *m*-by-*n* matrix which is zero except for its *n* diagonal elements, *U* is an *m*-by-*n* (or *m*-by-*m*) orthonormal matrix, and *V* is an *n*-by-*n* orthogonal matrix. The diagonal elements of Σ are the singular values of *A*; the columns of *U* and *V* are the left and right singular vectors of *A*, respectively. The matrices *U* and *V* are computed and stored in the arrays *u* and *v*, respectively. The diagonal of Σ is computed and stored in the array *sva*.

The ?gejsv routine can sometimes compute tiny singular values and their singular vectors much more accurately than other SVD routines.

The routine implements a preconditioned Jacobi SVD algorithm. It uses ?geqp3, ?geqrf, and ?gelqf as preprocessors and preconditioners. Optionally, an additional row pivoting can be used as a preprocessor, which in some cases results in much higher accuracy. An example is matrix *A* with the structure A = D1 * C * D2, where D1, D2 are arbitrarily ill-conditioned diagonal matrices and C is a well-conditioned matrix. In that case, complete pivoting in the first QR factorizations provides accuracy dependent on the condition number of C, and independent of D1, D2. Such higher accuracy is not completely understood theoretically, but it works well in practice.

If *A* can be written as `A = B*D`, with well-conditioned *B* and some diagonal *D*, then the high accuracy is guaranteed, both theoretically and in software, independent of *D*. For more details see [Drmac08-1], [Drmac08-2].

The computational range for the singular values can be the full range (`UNDERFLOW`,`OVERFLOW`), provided that the machine arithmetic and the BLAS and LAPACK routines called by `?gejsv` are implemented to work in that range. If that is not the case, the restriction for safe computation with the singular values in the range of normalized IEEE numbers is that the spectral condition number `kappa(A)=sigma_max(A)/sigma_min(A)` does not overflow. This code (`?gejsv`) is best used in this restricted range, meaning that singular values of magnitude below `||A||_2 / slamch('O')` (for single precision) or `||A||_2 / dlamch('O')` (for double precision) are returned as zeros. See *jobr* for details on this.

This implementation is slower than the one described in [Drmac08-1], [Drmac08-2] due to replacement of some non-LAPACK components, and because the choice of some tuning parameters in the iterative part (`?gesvj`) is left to the implementer on a particular machine.

The rank revealing QR factorization (in this code: `?geqp3`) should be implemented as in [Drmac08-3].

If *m* is much larger than *n*, it is obvious that the inital QRF with column pivoting can be preprocessed by the QRF without pivoting. That well known trick is not used in `?gejsv` because in some cases heavy row weighting can be treated with complete pivoting. The overhead in cases *m* much larger than *n* is then only due to pivoting, but the benefits in accuracy have prevailed. You can incorporate this extra QRF step easily and also improve data movement (matrix transpose, matrix copy, matrix transposed copy) - this implementation of `?gejsv` uses only the simplest, naive data movement.

Input Parameters

joba

`CHARACTER*1`. Must be `'C'`, `'E'`, `'F'`, `'G'`, `'A'`, or `'R'`.

Specifies the level of accuracy:

If *joba* `= 'C'`, high relative accuracy is achieved if *A* `= B*D` with well-conditioned *B* and arbitrary diagonal matrix *D*. The accuracy cannot be spoiled by column scaling. The accuracy of the computed output depends on the condition of *B*, and the procedure aims at the best theoretical accuracy. The relative error `max_{i=1:N}|d sigma_i| / sigma_i` is bounded by `f(M,N)*epsilon* cond(B)`, independent of *D*. The input matrix is preprocessed with the QRF with column pivoting. This initial preprocessing and preconditioning by a rank revealing QR factorization is common for all values of *joba*. Additional actions are specified as follows:

If *joba* `= 'E'`, computation as with `'C'` with an additional estimate of the condition number of *B*. It provides a realistic error bound.

If *joba* `= 'F'`, accuracy higher than in the `'C'` option is achieved, if *A* `= D1*C*D2` with ill-conditioned diagonal scalings *D1*, *D2*, and a well-conditioned matrix *C*. This option is advisable, if the structure of the input matrix is not known and relative accuracy is desirable. The input matrix *A* is preprocessed with QR factorization with full (row and column) pivoting.

If *joba* = 'G', computation as with 'F' with an additional estimate of the condition number of *B*, where *A* = *B***D*. If *A* has heavily weighted rows, using this condition number gives too pessimistic error bound.

If *joba* = 'A', small singular values are the noise and the matrix is treated as numerically rank defficient. The error in the computed singular values is bounded by f(m,n)*epsilon*||A||. The computed SVD A = U*S*V**t (for real flavors) or A = U*S*V**H (for complex flavors) restores *A* up to f(m,n)*epsilon*||A||. This enables the procedure to set all singular values below n*epsilon*||A|| to zero.

If *joba* = 'R', the procedure is similar to the 'A' option. Rank revealing property of the initial QR factorization is used to reveal (using triangular factor) a gap sigma_{r+1} < epsilon * sigma_r, in which case the numerical rank is declared to be *r*. The SVD is computed with absolute error bounds, but more accurately than with 'A'.

jobu CHARACTER*1. Must be 'U', 'F', 'W', or 'N'.

Specifies whether to compute the columns of the matrix *U*:

If *jobu* = 'U', *n* columns of *U* are returned in the array *u*

If *jobu* = 'F', a full set of *m* left singular vectors is returned in the array *u*.

If *jobu* = 'W', *u* may be used as workspace of length *m***n*. See the description of *u*.

If *jobu* = 'N', *u* is not computed.

jobv CHARACTER*1. Must be 'V', 'J', 'W', or 'N'.

Specifies whether to compute the matrix *V*:

If *jobv* = 'V', *n* columns of *V* are returned in the array *v*; Jacobi rotations are not explicitly accumulated.

If *jobv* = 'J', *n* columns of *V* are returned in the array *v* but they are computed as the product of Jacobi rotations. This option is allowed only if *jobu*≠'N'

If *jobv* = 'W', *v* may be used as workspace of length *n***n*. See the description of *v*.

If *jobv* = 'N', *v* is not computed.

jobr CHARACTER*1. Must be 'N' or 'R'.

Specifies the range for the singular values. If small positive singular values are outside the specified range, they may be set to zero. If *A* is scaled so that the largest singular value of the scaled matrix is around sqrt(big), big = ?lamch('O'), the function can remove columns of *A* whose norm in the scaled matrix is less than sqrt(?lamch('S')) (for *jobr* = 'R'), or less than small = ?lamch('S')/?lamch('E').

If *jobr* = 'N', the function does not remove small columns of the scaled matrix. This option assumes that BLAS and QR factorizations and triangular solvers are implemented to work in that range. If the condition of *A* if greater that big, use ?gesvj.

If *jobr* = 'R', restricted range for singular values of the scaled matrix *A* is [sqrt(?lamch('S'), sqrt(big)], roughly as described above. This option is recommended.

For computing the singular values in the full range [?lamch('S'),big], use ?gesvj.

jobt CHARACTER*1. Must be 'T' or 'N'.

If the matrix is square, the procedure may determine to use a transposed *A* if A^T (for real flavors) or A^H (for complex flavors) seems to be better with respect to convergence. If the matrix is not square, *jobt* is ignored.

The decision is based on two values of entropy over the adjoint orbit of A^T * *A* (for real flavors) or A^H * *A* (for complex flavors). See the descriptions of *work(6)* and *work(7)*.

If *jobt* = 'T', the function performs transposition if the entropy test indicates possibly faster convergence of the Jacobi process, if *A* is taken as input. If *A* is replaced with A^T or A^H, the row pivoting is included automatically.

If *jobt* = 'N', the functions attempts no speculations. This option can be used to compute only the singular values, or the full SVD (*u*, *sigma*, and *v*). For only one set of singular vectors (*u* or *v*), the caller should provide both *u* and *v*, as one of the arrays is used as workspace if the matrix *A* is transposed. The implementer can easily remove this constraint and make the code more complicated. See the descriptions of *u* and *v*.

Caution
The *jobt* = 'T' option is experimental and its effect might not be the same in subsequent releases. Consider using the *jobt* = 'N' instead.

jobp CHARACTER*1. Must be 'P' or 'N'.

Enables structured perturbations of denormalized numbers. This option should be active if the denormals are poorly implemented, causing slow computation, especially in cases of fast convergence. For details, see [Drmac08-1], [Drmac08-2] . For simplicity, such perturbations are included only when the full SVD or only the singular values are requested. You can add the perturbation for the cases of computing one set of singular vectors.

If *jobp* = 'P', the function introduces perturbation.

If *jobp* = 'N', the function introduces no perturbation.

m INTEGER. The number of rows of the input matrix *A*; $m \geq 0$.

n INTEGER. The number of columns in the input matrix *A*; $m \geq n \geq 0$.

a, *u*, *v* REAL for sgejsv

DOUBLE PRECISION for dgejsv.

COMPLEX for cgejsv

DOUBLE COMPLEX for zgejsv

Array *a*(*lda*,*n*) is an array containing the *m*-by-*n* matrix *A*.

u is a workspace array, its size is (*ldu*,*); the second dimension of *u* must be *m* if *jobu* = 'F' and *n* otherwise, . When *jobt* = 'T' and *m* = *n*, *u* must be provided even though *jobu* = 'N'.

v is a workspace array, its size is (*ldv*,*n*). When *jobt* = 'T' and *m* = *n*, *v* must be provided even though *jobv* = 'N'.

lda INTEGER. The leading dimension of the array *a*. Must be at least max(1, *m*) .

sva REAL for sgejsv

DOUBLE PRECISION for dgejsv.

REAL for cgejsv

DOUBLE PRECISION for zgejsv

sva is a workspace array, its size is *n*.

ldu INTEGER. The leading dimension of the array *u*; *ldu*≥ 1.

jobu = 'U' or 'F' or 'W', *ldu*≥*m* for column major layout.

ldv INTEGER. The leading dimension of the array *v*; *ldv*≥ 1.

jobv = 'V' or 'J' or 'W', *ldv*≥*n*.

work REAL for sgejsv

DOUBLE PRECISION for dgejsv.

work is a workspace array, its dimension max(7, *lwork*).

lwork INTEGER.

For real flavors:

Length of *work* to confirm proper allocation of work space. *lwork* depends on the task performed:

If only *sigma* is needed (*jobu* = 'N', *jobv* = 'N') and

- no scaled condition estimate is required, then *lwork*≥ max(2*m+n,4*n+1,7). This is the minimal requirement. For optimal performance (blocked code) the optimal value is *lwork*≥ max(2*m+n,3*n+(n+1)*nb,7). Here *nb* is the optimal block size for ?geqp3/?geqrf.

  In general, the optimal length *lwork* is computed as

  lwork≥ max(2*m+n,n+*lwork*(sgeqp3),n+*lwork*(sgeqrf),7) for sgejsv

  lwork≥ max(2*m+n,n+*lwork*(dgeqp3),n+*lwork*(dgeqrf),7) for dgejsv
- ... an estimate of the scaled condition number of *A* is required (*joba* = 'E', 'G'). In this case, *lwork* is the maximum of the above and *n*n+4*n*, that is, *lwork*≥ max(2*m+n,n*n+4*n,7). For optimal performance (blocked code) the optimal value is *lwork*≥ max(2*m+n,3*n+(n+1)*nb, n*n+4*n, 7).

In general, the optimal length *lwork* is computed as

lwork≥ max(2*m*+*n*,*n*+*lwork*(sgeqp3),*n*+*lwork*(sgeqrf),*n*+*n***n* +*lwork*(spocon, 7) for sgejsv

lwork≥ max(2*m*+*n*,*n*+*lwork*(dgeqp3),*n*+*lwork*(dgeqrf),*n*+*n***n* +*lwork*(dpocon, 7) for dgejsv

If *sigma* and the right singular vectors are needed (*jobv* = 'V'),

- the minimal requirement is *lwork*≥ max(2*m*+*n*,4*n*+1,7).
- for optimal performance, *lwork*≥ max(2*m*+*n*,3*n*+(*n*+1)**nb*,7), where *nb* is the optimal block size for ?geqp3, ?geqrf, ?gelqf, ?ormlq. In general, the optimal length *lwork* is computed as

  lwork≥ max(2*m*+*n*, *n*+*lwork*(sgeqp3), *n*+*lwork*(spocon), *n* +*lwork*(sgelqf), 2*n*+*lwork*(sgeqrf), *n*+*lwork*(sormlq) for sgejsv

  lwork≥ max(2*m*+*n*, *n*+*lwork*(dgeqp3), *n*+*lwork*(dpocon), *n* +*lwork*(dgelqf), 2*n*+*lwork*(dgeqrf), *n*+*lwork*(dormlq) for dgejsv

If *sigma* and the left singular vectors are needed

- the minimal requirement is *lwork*≥ max(2*n*+*m*,4*n*+1,7).
- for optimal performance,

  if *jobu* = 'U' :: *lwork*≥ max(2*m*+*n*,3*n*+(*n*+1)**nb*, 7),

  if *jobu* = 'F' :: *lwork*≥ max(2*m*+*n*,3*n*+(*n*+1)**nb*, *n*+*m***nb*, 7),

  where *nb* is the optimal block size for ?geqp3, ?geqrf, ?ormlq . In general, the optimal length *lwork* is computed as

  lwork≥ max(2*m*+*n*, *n*+*lwork*(sgeqp3), *n*+*lwork*(spocon), 2**n* +*lwork*(sgeqrf), *n*+*lwork*(sormlq) for sgejsv

  lwork≥ max(2*m*+*n*, *n*+*lwork*(dgeqp3), *n*+*lwork*(dpocon), 2**n* +*lwork*(dgeqrf), *n*+*lwork*(dormlq) for dgejsv

  Here *lwork*(?ormlq) equals *n***nb* (for *jobu* = 'U') or *m***nb* (for *jobu* = 'F')

If full SVD is needed (*jobu* = 'U' or 'F') and

- if *jobv* = 'V',

  the minimal requirement is *lwork*≥ max(2*m*+*n*, 6*n*+2*n***n*)
- if *jobv* = 'J',

  the minimal requirement is *lwork*≥ max(2*m*+*n*, 4*n*+*n***n*, 2*n*+*n***n* +6)
- For optimal performance, *lwork* should be additionally larger than *n* +*m***nb*, where *nb* is the optimal block size for ?ormlq.

For complex flavors:

Length of *cwork* to confirm proper allocation of workspace. The value of *lwork* depends on the job:

- If only *sigma* is needed (*jobu*.EQ.'N', *jobv*.EQ.'N') and

- no scaled condition estimate is required: *lwork*≥ 2*n+1. This is the minimal requirement. For optimal performance (blocked code) the optimal value is *lwork*≥*n* + (*n*+1)*nb*. Here *nb* is the optimal block size for ?geqp3 and ?geqrf. In general, optimal *lwork* is computed as *lwork*≥ max(*n*+*lwork*(?geqp3), *n*+*lwork*(?geqrf), *lwork*(?gesvj)).
- an estimate of the scaled condition number of *a* is required (*joba*='E' or 'G'). In this case, the minimal requirement is *lwork*≥*n***n* + 2**n*. For optimal performance (blocked code) the optimal value is *lwork*≥ max(*n*+(*n*+1)**nb*, *n***n*+3**n*) = n^2 + 2**n*. In general, the optimal length *lwork* is computed as *lwork*≥ max(*n* +*lwork*(?geqp3),*n*+*lwork*(?geqrf), *lwork*(?gesvj), *n*+*n***n* +*lwork*(?pocon)).

- If *sigma* and the right singular vectors are needed (*jobv*.EQ.'V' or *jobu*.EQ.'N') and
  - no scaled condition estimate is requested (*jobe* .EQ.'N'), then the minimal requirement is *lwork*≥ 3**n*. For optimal performance, *lwork*≥ max(*n*+(*n*+1)**nb*, 2**n*+*n***nb*) = 2**n*+*n***nb*, where *nb* is the optimal block size for ?geqp3, ?geqrf, ?gelq, ?unmlq.

    In general, the optimal length *lwork* is computed as *lwork*≥ max(*n* +*lwork*(?geqp3), *n*+*lwork*(?gesvj), *n*+*lwork*(?gelqf), 2**n* +*lwork*(?geqrf), *n*+*lwork*(?unmlq)).
  - an estimate of the scaled condition number of *a* is required (*joba*='E' or 'G'), then the minimal requirement is *lwork*≥ 3**n*. For optimal performance, *lwork*≥ max(*n*+(*n*+1)**nb*, 2**n*, 2**n* +*n***nb*) = 2**n*+*n***nb*, where *nb* is the optimal block size for ?geqp3, ?geqrf, ?gelq, ?unmlq

    In general, the optimal length *lwork* is computed as *lwork*≥ max(*n* +*lwork*(?geqp3), *n*+*lwork*(?pocon), *n*+*lwork*(?gesvj), *n* +*lwork*(?gelqf), 2**n*+*lwork*(?geqrf), *n*+*lwork*(?unmlq))
- If *sigma* and the left singular vectors are needed and
  - no scaled condition estimate is requested (*jobe* .EQ.'N'), then the minimal requirement is *lwork*≥ 3**n*.

    For optimal performance: if *jobu*.EQ.'U' :: *lwork*≥ max(3**n*, *n* +(*n*+1)**nb*, 2**n*+*n***nb*) = 2**n*+*n***nb*, where *nb* is the optimal block size for ?geqp3, ?geqrf, ?unmqr. In general, the optimal length *lwork* is computed as *lwork*≥ max(*n*+*lwork*(?geqp3),2**n* +*lwork*(?geqrf), *n*+*lwork*(?unmqr)).
  - an estimate of the scaled condition number of *a* is required (*joba*='E' or 'G'), then the minimal requirement is *lwork*≥ 3**n*.

    For optimal performance: if *jobu*.EQ.'U' :: *lwork*≥ max(3**n*, *n* +(*n*+1)**nb*, 2**n*+*n***nb*) = 2**n*+*n***nb*, where *nb* is the optimal block size for ?geqp3, ?geqrf, ?unmqr. In general, the optimal length *lwork* is computed as *lwork*≥ max(*n*+*lwork*(?geqp3),*n* +*lwork*(?pocon), 2**n*+*lwork*(?geqrf), *n*+*lwork*(?unmqr)).

then the minimal requirement is *lwork*≥ 3**n*. For optimal performance: if *jobu*.EQ.'U' :: *lwork*≥ max(3**n*, *n*+(*n*+1)**nb*, 2**n*+*n***nb*), where *nb* is the optimal block size for ?geqp3, ?geqrf, ?unmqr. In general, the optimal length *lwork* is computed as *lwork*≥ max(*n* +*lwork*(?geqp3),*n*+*lwork*(?pocon), 2**n*+*lwork*(?geqrf), *n* +*lwork*(?unmqr)).

- If the full SVD is needed: (*jobu*.EQ.'U' or *jobu*.EQ.'F') and
  - if *jobv*.EQ.'V' the minimal requirement is *lwork*≥ 5**n*+2**n***n*.
  - if *jobv*.EQ.'J' the minimal requirement is *lwork*≥ 4**n*+*n***n*. In both cases, the allocated *cwork* can accommodate blocked runs of ?geqp3, ?geqrf, ?gelqf, ?unmqr, ?unmlq.

If the call to ?gejsv is a workspace query (indicated by *lwork* = -1 or *lrwork* = -1), then on exit *cwork*(1) contains the required length of *cwork* for the job parameters used in the call.

cwork

COMPLEX for cgejsv

DOUBLE COMPLEX for zgejsv

cwork is a workspace array of size max(2, *lwork*).

If the call to ?gejsv is a workspace query (indicated by *lwork* = -1 or *lrwork* = -1), then on exit *cwork*(1) contains the required length of *cwork* for the job parameters used in the call.

rwork

REAL for cgejsv

DOUBLE PRECISION for zgejsv

rwork is an array of size at least max(7, *lrwork*) for real flavors and at least max(7, *lwork*) for complex flavors.

lrwork

INTEGER. Length of *rwork* to confirm proper allocation of workspace. *lrwork* depends on the job:

1. If only singular values are requested i.e. if lsame(*jobu*,'N') .AND. lsame(*jobv*,'N') then:

- If lsame(*jobt*,'T') .OR. lsame(*joba*,'F') .OR. lsame(*joba*,'G'), then *lrwork* = max(7, 2 * *m*).
- Otherwise, *lrwork* = max(7, *n*).

2. If singular values with the right singular vectors are requested i.e. if (lsame(*jobv*,'V').OR.lsame(*jobv*,'J')) .AND. .NOT. (lsame(*jobu*,'U').OR.lsame(*jobu*,'F')) then:

- If lsame(*jobt*,'T') .OR. lsame(*joba*,'F') .OR. lsame(*joba*,'G'), then *lrwork* = max(7, 2 * *m*).
- Otherwise, *lrwork* = max(7, *n*).

3. If singular values with the left singular vectors are requested, i.e. if (lsame(*jobu*,'U').OR.lsame(*jobu*,'F')) .AND. .NOT. (lsame(*jobv*,'V').OR.lsame(*jobv*,'J')) then:

- If lsame(*jobt*,'T') .OR. lsame(*joba*,'F') .OR. lsame(*joba*,'G'), then *lrwork* = max(7, 2 * *m*).
- Otherwise, *lrwork* = max(7, *n*).

4. If singular values with both the left and the right singular vectors are requested, i.e. if `(lsame(jobu,'U').OR.lsame(jobu,'F')) .AND. (lsame(jobv,'V').OR.lsame(jobv,'J'))` then:

- If `lsame(jobt,'T') .OR. lsame(joba,'F') .OR. lsame(joba,'G')`, then `lrwork = max( 7, 2 * m )`.
- Otherwise, `lrwork = max( 7, n )`.

For complex flavors, if the call to `?gejsv` is a workspace query (indicated by `lwork` = -1 or `lrwork` = -1), then on exit `rwork(1)` contains the required length of `rwork` for the job parameters used in the call.

iwork

`INTEGER`. Workspace array, of size

For real flavors:

).

max(3, *m*+3**n*max(3, 2 * *n* + *m*).

For complex flavors, the size depends on the job but is at least 4:

- If only the singular values are requested and `lsame(jobt,'T') .OR. lsame(joba,'F') .OR. lsame(joba,'G')`, then the length of *iwork* is *n* + *m*; otherwise the length of *iwork* is *n*.
- If the singular values and the right singular vectors are requested and `lsame(jobt,'T') .OR. lsame(joba,'F') .OR. lsame(joba,'G')`, then the length of *iwork* is *n* + *m*; otherwise the length of *iwork* is *n*.
- If the singular values and the left singular vectors are requested and `lsame(jobt,'T') .OR. lsame(joba,'F') .OR. lsame(joba,'G')`, then the length of *iwork* is *n* + *m*; otherwise the length of *iwork* is *n*.
- If the singular values and both the left and the right singular vectors are requested and
  - if `lsame(jobv,'J')`, if `lsame(jobt,'T') .OR. lsame(joba,'F') .OR. lsame(joba,'G')`, then the length of *iwork* is *n* + *m*; otherwise the length of *iwork* is *n*.
  - if `lsame(jobv,'V')` , if `lsame(jobt,'T') .OR. lsame(joba,'F') .OR. lsame(joba,'G')`, then the length of *iwork* is 2**n* + *m*; otherwise the length of *iwork* is 2**n*.

  and `lsame(jobt,'T') .OR. lsame(joba,'F') .OR. lsame(joba,'G')`, then the length of *iwork* is *n* + *m*; otherwise the length of *iwork* is *n*.

Output Parameters

sva

On exit:

For `work(1)/work(2) = one`: the singular values of *A*. During the computation *sva* contains Euclidean column norms of the iterated matrices in the array *a*.

For *work(1)≠work(2)*: the singular values of *A* are `(work(1)/work(2)) * sva(1:n)`. This factored form is used if *sigma_max(A)* overflows or if small singular values have been saved from underflow by scaling the input matrix *A*.

jobr = 'R', some of the singular values may be returned as exact zeros obtained by 'setting to zero' because they are below the numerical rank threshold or are denormalized numbers.

u On exit:

If *jobu* = 'U', contains the *m*-by-*n* matrix of the left singular vectors.

If *jobu* = 'F', contains the *m*-by-*m* matrix of the left singular vectors, including an orthonormal basis of the orthogonal complement of the range of *A*.

If *jobu* = 'W' and *jobv* = 'V', *jobt* = 'T', and *m* = *n*, then *u* is used as workspace if the procedure replaces *A* with A^T (for real flavors) or A^H (for complex flavors). In that case, *v* is computed in *u* as left singular vectors of A^T or A^H and copied back to the *v* array. This 'W' option is just a reminder to the caller that in this case *u* is reserved as workspace of length *n***n*.

If *jobu* = 'N', *u* is not referenced.

v On exit:

If *jobv* = 'V' or 'J', contains the *n*-by-*n* matrix of the right singular vectors.

If *jobv* = 'W' and *jobu* = 'U', *jobt* = 'T', and *m* = *n*, then *v* is used as workspace if the procedure replaces *A* with A^T (for real flavors) or A^H (for complex flavors). In that case, *u* is computed in *v* as right singular vectors of A^T or A^H and copied back to the *u* array. This 'W' option is just a reminder to the caller that in this case *v* is reserved as workspace of length *n***n*.

If *jobv* = 'N', *v* is not referenced.

work On exit,

work(1) = *scale* = *work(2)*/*work(1)* is the scaling factor such that *scale***sva*(1:*n*) are the computed singular values of *A*. See the description of *sva()*.

work(2) = see the description of *work(1)*.

work(3) = *sconda* is an estimate for the condition number of column equilibrated *A*. If *joba* = 'E' or 'G', *sconda* is an estimate of sqrt(|| (R**t * R)**(-1)||_1). It is computed using ?pocon. It holds *n***(-1/4) * *sconda*≤ ||R**(-1)||_2 ≤*n***(1/4) * *sconda*, where R is the triangular factor from the QRF of *A*. However, if R is truncated and the numerical rank is determined to be strictly smaller than *n*, *sconda* is returned as -1, indicating that the smallest singular values might be lost.

If full SVD is needed, the following two condition numbers are useful for the analysis of the algorithm. They are provided for a user who is familiar with the details of the method.

work(4) = an estimate of the scaled condition number of the triangular factor in the first QR factorization.

work(5) = an estimate of the scaled condition number of the triangular factor in the second QR factorization.

The following two parameters are computed if *jobt* = 'T'. They are provided for a user who is familiar with the details of the method.

work(6) = the entropy of *A**t**A* :: this is the Shannon entropy of diag(*A**t**A*) / Trace(*A**t**A*) taken as point in the probability simplex.

work(7) = the entropy of *A***A**t.

rwork

On exit,

rwork(1) determines the scaling factor *scale* = *rwork(2)* / *rwork(1)* such that *scale***sva*(1:*n*) are the computed singular values of *a*. (See the description of *sva()*.)

rwork(2) = see the description of *rwork(1)*.

rwork(3) = *sconda* is an estimate for the condition number of column equilibrated *A*. If *joba* = 'E' or 'G', *sconda* is an estimate of SQRT(||(R^* * R)^(-1)||_1). It is computed using ?pocon. It holds *n*^(-1/4) * *sconda*≤ ||R^(-1)||_2 ≤*n*^(1/4) * *sconda* where R is the triangular factor from the QRF of *A*. However, if R is truncated and the numerical rank is determined to be strictly smaller than *n*, *sconda* is returned as -1, thus indicating that the smallest singular values might be lost.

If full SVD is needed, the following two condition numbers are useful for the analysis of the algorithm. They are provided for a user who is familiar with the details of the method.

rwork(4) = an estimate of the scaled condition number of the triangular factor in the first QR factorization.

rwork(5) = an estimate of the scaled condition number of the triangular factor in the second QR factorization.

The following two parameters are computed if *jobt* = 'T'. They are provided for a user who is familiar with the details of the method.

rwork(6) = the entropy of *A*^* * *A* :: this is the Shannon entropy of diag(*A*^* * *A*) / Trace(*A*^* * *A*) taken as point in the probability simplex.

rwork(7) = the entropy of *A* * *A*^*. (See the description of *rwork(6)*.)

For complex flavors, if the call to ?gejsv is a workspace query (indicated by *lwork* = -1 or *lrwork* = -1), then on exit *rwork*(1) contains the required length of *rwork* for the job parameters used in the call.

iwork

INTEGER. On exit,

iwork(1) = the numerical rank determined after the initial QR factorization with pivoting. See the descriptions of *joba* and *jobr*.

iwork(2) = the number of the computed nonzero singular value.

iwork(3) = if nonzero, a warning message. If *iwork(3)*=1, some of the column norms of *A* were denormalized floats. The requested high accuracy is not warranted by the data.

For complex flavors, *iwork(4)* = 1 or -1. If *iwork(4)* = 1, then the procedure used A^H to do the job as specified by the *job* parameters.

For complex flavors, if the call to ?gejsv is a workspace query (indicated by *lwork* = -1 or *lrwork* = -1), then on exit *iwork*(1) contains the required length of *iwork* for the job parameters used in the call.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, the function did not converge in the maximal number of sweeps. The computed values may be inaccurate.

See Also

?geqp3
?geqrf
?gelqf
?gesvj
?lamch
?pocon
?ormlq

?gesvj

Computes the singular value decomposition of a real matrix using Jacobi plane rotations.

Syntax

```
call sgesvj(joba, jobu, jobv, m, n, a, lda, sva, mv, v, ldv, work, lwork, info)
call dgesvj(joba, jobu, jobv, m, n, a, lda, sva, mv, v, ldv, work, lwork, info)
call cgesvj(joba, jobu, jobv, m, n, a, lda, sva, mv, v, ldv, cwork, lwork, rwork, lrwork, info )
call zgesvj(joba, jobu, jobv, m, n, a, lda, sva, mv, v, ldv, cwork, lwork, rwork, lrwork, info )
```

Include Files

- mkl.fi

Description

The routine computes the singular value decomposition (SVD) of a real or complex m-by-n matrix A, where $m \geq n$.

The SVD of A is written as

$A = U * \Sigma * V^T$ for real flavors, or

$A = U * \Sigma * V^H$ for complex flavors,

where Σ is an m-by-n diagonal matrix, U is an m-by-n orthonormal matrix, and V is an n-by-n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A; the columns of U and V are the left and right singular vectors of A, respectively. The matrices U and V are computed and stored in the arrays u and v, respectively. The diagonal of Σ is computed and stored in the array *sva*.

The ?gesvj routine can sometimes compute tiny singular values and their singular vectors much more accurately than other SVD routines.

The n-by-n orthogonal matrix V is obtained as a product of Jacobi plane rotations. The rotations are implemented as fast scaled rotations of Anda and Park [AndaPark94]. In the case of underflow of the Jacobi angle, a modified Jacobi transformation of Drmac ([Drmac08-4]) is used. Pivot strategy uses column interchanges of de Rijk ([deRijk98]). The relative accuracy of the computed singular values and the accuracy of the computed singular vectors (in angle metric) is as guaranteed by the theory of Demmel and Veselic [Demmel92]. The condition number that determines the accuracy in the full rank case is essentially

$$\left(\min_{i} d_{ii}\right) \cdot \kappa(A \cdot D)$$

where $\kappa(.)$ is the spectral condition number. The best performance of this Jacobi SVD procedure is achieved if used in an accelerated version of Drmac and Veselic [Drmac08-1], [Drmac08-2].

The computational range for the nonzero singular values is the machine number interval (`UNDERFLOW`,`OVERFLOW`). In extreme cases, even denormalized singular values can be computed with the corresponding gradual loss of accurate digit.

Input Parameters

joba

`CHARACTER*1`. Must be `'L'`, `'U'` or `'G'`.

Specifies the structure of *A*:

If *joba* = `'L'`, the input matrix *A* is lower triangular.

If *joba* = `'U'`, the input matrix *A* is upper triangular.

If *joba* = `'G'`, the input matrix *A* is a general m-by-n, $m \geq n$.

jobu

`CHARACTER*1`. Must be `'U'`, `'C'` or `'N'`.

Specifies whether to compute the left singular vectors (columns of *U*):

If *jobu* = `'U'`, the left singular vectors corresponding to the nonzero singular values are computed and returned in the leading columns of *A*. See more details in the description of *a*. The default numerical orthogonality threshold is set to approximately `TOL=CTOL*EPS, CTOL=sqrt(m), EPS = ?lamch('E')`

If *jobu* = `'C'`, analogous to *jobu* = `'U'`, except that you can control the level of numerical orthogonality of the computed left singular vectors. `TOL` can be set to `TOL=CTOL*EPS`, where `CTOL` is given on input in the array *work*. No `CTOL` smaller than ONE is allowed. `CTOL` greater than `1 / EPS` is meaningless. The option `'C'` can be used if *m*`*EPS` is satisfactory orthogonality of the computed left singular vectors, so `CTOL=`*m* could save a few sweeps of Jacobi rotations. See the descriptions of *a* and *work(1)*.

If *jobu* = `'N'`, *u* is not computed. However, see the description of *a*.

jobv

`CHARACTER*1`. Must be `'V'`, `'A'` or `'N'`.

Specifies whether to compute the right singular vectors, that is, the matrix *V*:

If *jobv* = `'V'`, the matrix *V* is computed and returned in the array *v*.

If *jobv* = 'A', the Jacobi rotations are applied to the *mv*-by-*n* array *v*. In other words, the right singular vector matrix *V* is not computed explicitly, instead it is applied to an *mv*-by-*n* matrix initially stored in the first *mv* rows of *V*.

If *jobv* = 'N', the matrix *V* is not computed and the array *v* is not referenced.

m

INTEGER. The number of rows of the input matrix *A*.

1/slamch('E')> *m*≥ 0 for sgesvj.

1/dlamch('E')> *m*≥ 0 for dgesvj.

n

INTEGER. The number of columns in the input matrix *A*; *m*≥*n*≥ 0.

a, *v*

REAL for sgesvj

DOUBLE PRECISION for dgesvj.

COMPLEX for cgesvj

DOUBLE COMPLEX for zgesvj

Array *a*(*lda*,*n*) is an array containing the *m*-by-*n* matrix *A*.

Array *v* is a workspace array, its dimension is (*ldv*,*); the second dimension of *v* must be at least max(1, *n*).

lda

INTEGER. The leading dimension of the array *a*. Must be at least max(1, *m*) .

mv

INTEGER.

If*jobv* = 'A', the product of Jacobi rotations in ?gesvj is applied to the first *mv* rows of *v*. See the description of *jobv*. 0 ≤*mv*≤*ldv*.

ldv

INTEGER. The leading dimension of the array *v*; *ldv*≥ 1.

jobv = 'V', *ldv*≥ max(1, *n*).

jobv = 'A', *ldv*≥ max(1, *mv*).

work

REAL for sgesvj

DOUBLE PRECISION for dgesvj.

work is a workspace array, its dimension max(4, *m*+*n*).

If *jobu* = 'C', *work(1)*=CTOL, where CTOL defines the threshold for convergence. The process stops if all columns of *A* are mutually orthogonal up to CTOL*EPS, EPS=?lamch('E'). It is required that CTOL ≥ 1, that is, it is not allowed to force the routine to obtain orthogonality below ε.

cwork

COMPLEX for cgesvj

DOUBLE COMPLEX for zgesvj

cwork is a workspace array, its dimension *m*+*n*.

lwork

INTEGER.

Length of *work* for real flavors or *cwork* for complex flavors, *lwork*≥ max(6,*m*+*n*).

rwork

REAL for cgesvj

DOUBLE PRECISION for zgesvj

rwork is a workspace array, its dimension max(6,*m*+*n*).

If *jobu* = 'C', *rwork(1)* = CTOL, where CTOL defines the threshold for convergence. The process stops if all columns of *a* are mutually orthogonal up to CTOL*EPS, EPS=?lamch('E'). It is required that CTOL≥ 1, that is, it is not allowed to force the routine to obtain orthogonality below ε.

lrwork

INTEGER for cgesvj

INTEGER for zgesvj

Length of *rwork*, *lrwork*≥ max(6,*n*).

Output Parameters

a

On exit:

If *jobu* = 'U' or *jobu* = 'C':

- if *info* = 0, the leading columns of *A* contain left singular vectors corresponding to the computed singular values of *a* that are above the underflow threshold ?lamch('S'), that is, non-zero singular values. The number of the computed non-zero singular values is returned in *work(2)* for real flavors or *rwork(2)* for complex flavors. Also see the descriptions of *sva* and *work* for real flavors or *rwork* for complex flavors. The computed columns of *u* are mutually numerically orthogonal up to approximately TOL=sqrt(*m*)*EPS (default); or TOL=CTOL*EPS*jobu* = 'C', see the description of *jobu*.
- if *info* > 0, the procedure ?gesvj did not converge in the given number of iterations (sweeps). In that case, the computed columns of *u* may not be orthogonal up to TOL. The output *u* (stored in *a*), *sigma* (given by the computed singular values in *sva*(1:n)) and *v* is still a decomposition of the input matrix *A* in the sense that the residual $||A-scale*U*sigma*V^{T}||_2 / ||A||_2$ for real flavors or $||A-scale*U*sigma*V^{H}||_2 / ||A||_2$ for complex flavors (where scale = *stat*[0]) is small.

If *jobu* = 'N':

- if *info* = 0, note that the left singular vectors are 'for free' in the one-sided Jacobi SVD algorithm. However, if only the singular values are needed, the level of numerical orthogonality of *u* is not an issue and iterations are stopped when the columns of the iterated matrix are numerically orthogonal up to approximately *m**EPS. Thus, on exit, *a* contains the columns of *u* scaled with the corresponding singular values.
- if *info* > 0, the procedure ?gesvj did not converge in the given number of iterations (sweeps).

sva

REAL for sgesvj

DOUBLE PRECISION for dgesvj.

REAL for cgesvj

DOUBLE PRECISION for zgesvj

Array size *n*.

If *info* = 0, depending on the value *scale* =*work(1)* for real flavors or *rwork(1)* for complex flavors, where *scale* is the scaling factor:

- if *scale* = 1, *sva*(1:n) contains the computed singular values of *a*.

  During the computation, *sva* contains the Euclidean column norms of the iterated matrices in the array *a*.
- if *scale*≠ 1, the singular values of *a* are *scale***sva*(1:n), and this factored representation is due to the fact that some of the singular values of *a* might underflow or overflow.

If *info* > 0, the procedure ?gesvj did not converge in the given number of iterations (sweeps) and *scale***sva*(1:n) may not be accurate.

v

On exit:

If *jobv* = 'V', contains the *n*-by-*n* matrix of the right singular vectors.

If *jobv* = 'A', then *v* contains the product of the computed right singular vector matrix and the initial matrix in the array *v*.

If *jobv* = 'N', *v* is not referenced.

work

On exit,

work(1) = *scale* is the scaling factor such that *scale***sva*(1:*n*) are the computed singular values of *A*. See the description of *sva*().

work(2) is the number of the computed nonzero singular values.

work(3) is the number of the computed singular values that are larger than the underflow threshold.

work(4) is the number of sweeps of Jacobi rotations needed for numerical convergence.

work(5) = max_{i≠j} |COS(A(:,i),A(:,j))| in the last sweep. This is useful information in cases when ?gesvj did not converge, as it can be used to estimate whether the output is still useful and for post festum analysis.

work(6) is the largest absolute value over all sines of the Jacobi rotation angles in the last sweep. It can be useful in a post festum analysis.

rwork

On exit,

rwork(1) = *scale* is the scaling factor such that *scale***sva*(1:*n*) are the computed singular values of *A*. See description of *sva()*.

rwork(2) is the number of the computed nonzero singular values.

rwork(3) is the number of the computed singular values that are larger than the underflow threshold.

rwork(4) is the number of sweeps of Jacobi rotations needed for numerical convergence.

rwork(5) = max_{i≠j} |COS(A(:,i),A(:,j))| in the last sweep. This is useful information in cases when ?gesvj did not converge, as it can be used to estimate whether the output is still useful and for post festum analysis.

rwork(6) is the largest absolute value over all sines of the Jacobi rotation angles in the last sweep. It can be useful for a post festum analysis.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, the function did not converge in the maximal number (30) of sweeps. The output may still be useful. See the description of *work* or *rwork*.

See Also

?lamch

?ggsvd
Computes the generalized singular value decomposition of a pair of general rectangular matrices (deprecated).

Syntax

```
call sggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v, ldv, q, ldq, work, iwork, info)
call dggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v, ldv, q, ldq, work, iwork, info)
call cggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v, ldv, q, ldq, work, rwork, iwork, info)
call zggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v, ldv, q, ldq, work, rwork, iwork, info)
call ggsvd(a, b, alpha, beta [, k] [,l] [,u] [,v] [,q] [,iwork] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

This routine is deprecated; use ggsvd3.

The routine computes the generalized singular value decomposition (GSVD) of an *m*-by-*n* real/complex matrix *A* and *p*-by-*n* real/complex matrix *B*:

$U'*A*Q = D_1*(0\ R), V'*B*Q = D_2*(0\ R),$

where *U*, *V* and *Q* are orthogonal/unitary matrices and *U*', *V*' mean transpose/conjugate transpose of *U* and *V* respectively.

Let *k*+*l* = the effective numerical rank of the matrix (*A*', *B*')', then *R* is a (*k*+*l*)-by-(*k*+*l*) nonsingular upper triangular matrix, D_1 and D_2 are *m*-by-(*k*+*l*) and *p*-by-(*k*+*l*) "diagonal" matrices and of the following structures, respectively:

If *m*-*k*-*l*≥0,

$$D_1 = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{array}$$

$$(0 \quad R) = \begin{array}{c} \\ k \\ l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{array},$$

where

C = diag($alpha(K+1)$,..., $alpha(K+l)$)

S = diag($beta(K+1)$,...,$beta(K+l)$)

$C^2 + S^2 = I$

R is stored in $a(1{:}k+l, n-k-l+1{:}n)$ on exit.

If $m-k-l < 0$,

$$D_1 = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{r} \\ m-k \\ k+l-m \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$(0 \quad R) = \begin{array}{r} \\ k \\ m-k \\ k+l-m \end{array} \begin{array}{c} \begin{array}{cccc} n-k-l & k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{array}$$

where

C = diag($alpha(K+1)$,..., $alpha(m)$),

S = diag($beta(K+1)$,...,$beta(m)$),

$C_2 + S_2 = I$

On exit,

$$\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$$

is stored in $a(1{:}m, n{-}k{-}l{+}1{:}n)$ and R_{33} is stored in $b(m{-}k{+}1{:}l, n{+}m{-}k{-}l{+}1{:}n)$.

The routine computes *C*, *S*, *R*, and optionally the orthogonal/unitary transformation matrices *U*, *V* and *Q*.

In particular, if *B* is an *n*-by-*n* nonsingular matrix, then the GSVD of *A* and *B* implicitly gives the SVD of $A*B^{-1}$:

$A*B^{-1} = U*(D_1*D_2^{-1})*V'$.

If (*A*', *B*')' has orthonormal columns, then the GSVD of *A* and *B* is also equal to the CS decomposition of *A* and *B*. Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$A'*A*x = \lambda*B'*B*x$.

Input Parameters

jobu — CHARACTER*1. Must be 'U' or 'N'.
If *jobu* = 'U', orthogonal/unitary matrix *U* is computed.
If *jobu* = 'N', *U* is not computed.

jobv — CHARACTER*1. Must be 'V' or 'N'.
If *jobv* = 'V', orthogonal/unitary matrix *V* is computed.
If *jobv* = 'N', *V* is not computed.

jobq — CHARACTER*1. Must be 'Q' or 'N'.
If *jobq* = 'Q', orthogonal/unitary matrix *Q* is computed.
If *jobq* = 'N', *Q* is not computed.

m — INTEGER. The number of rows of the matrix *A* (*m*≥ 0).

n — INTEGER. The number of columns of the matrices *A* and *B* (*n*≥ 0).

p — INTEGER. The number of rows of the matrix *B* (*p*≥ 0).

a, *b*, *work* — REAL for sggsvd
DOUBLE PRECISION for dggsvd
COMPLEX for cggsvd
DOUBLE COMPLEX for zggsvd.
Arrays:
a(*lda*,*) contains the *m*-by-*n* matrix *A*.
The second dimension of *a* must be at least max(1, *n*).
b(*ldb*,*) contains the *p*-by-*n* matrix *B*.
The second dimension of *b* must be at least max(1, *n*).
work(*) is a workspace array.
The dimension of *work* must be at least max(3*n*, *m*, *p*)+*n*.

lda — INTEGER. The leading dimension of *a*; at least max(1, *m*).

ldb — INTEGER. The leading dimension of *b*; at least max(1, *p*).

ldu — INTEGER. The leading dimension of the array *u* .
ldu≥ max(1, *m*) if *jobu* = 'U'; *ldu*≥ 1 otherwise.

ldv — INTEGER. The leading dimension of the array *v* .
ldv≥ max(1, *p*) if *jobv* = 'V'; *ldv*≥ 1 otherwise.

ldq — INTEGER. The leading dimension of the array *q* .
ldq≥ max(1, *n*) if *jobq* = 'Q'; *ldq*≥ 1 otherwise.

iwork — INTEGER.
Workspace array, size at least max(1, *n*).

rwork

REAL for cggsvd DOUBLE PRECISION for zggsvd.

Workspace array, size at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

k, *l*

INTEGER. On exit, *k* and *l* specify the dimension of the subblocks. The sum *k*+*l* is equal to the effective numerical rank of (*A*', *B*')'.

a

On exit, *a* contains the triangular matrix *R* or part of *R*.

b

On exit, *b* contains part of the triangular matrix R if *m*-*k*-*l* < 0.

alpha, *beta*

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays, size at least max(1, *n*) each.

Contain the generalized singular value pairs of *A* and *B*:

alpha(1:*k*) = 1,

beta(1:*k*) = 0,

and if *m*-*k*-*l*≥ 0,

alpha(*k*+1:*k*+*l*) = *C*,

beta(*k*+1:*k*+*l*) = *S*,

or if *m*-*k*-*l* < 0,

alpha(*k*+1:*m*)= *C*, *alpha*(*m*+1:*k*+*l*)=0

beta(*k*+1:*m*) = *S*, *beta*(*m*+1:*k*+*l*) = 1

and

alpha(*k*+*l*+1:*n*) = 0

beta(*k*+*l*+1:*n*) = 0.

u, *v*, *q*

REAL for sggsvd

DOUBLE PRECISION for dggsvd

COMPLEX for cggsvd

DOUBLE COMPLEX for zggsvd.

Arrays:

u(*ldu*,*); the second dimension of *u* must be at least max(1, *m*).

If *jobu* = 'U', *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*.

If *jobu* = 'N', *u* is not referenced.

v(*ldv*,*); the second dimension of *v* must be at least max(1, *p*).

If *jobv* = 'V', *v* contains the *p*-by-*p* orthogonal/unitary matrix *V*.

If *jobv* = 'N', *v* is not referenced.

q(*ldq*,*); the second dimension of *q* must be at least max(1, *n*).

If *jobq* = 'Q', *q* contains the *n*-by-*n* orthogonal/unitary matrix *Q*.

If *jobq* = 'N', *q* is not referenced.

iwork — On exit, *iwork* stores the sorting information.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, the Jacobi-type procedure failed to converge. For further details, see subroutine tgsja.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ggsvd interface are the following:

a — Holds the matrix *A* of size (*m*, *n*).

b — Holds the matrix *B* of size (*p*, *n*).

alpha — Holds the vector of length *n*.

beta — Holds the vector of length *n*.

u — Holds the matrix *U* of size (*m*, *m*).

v — Holds the matrix *V* of size (*p*, *p*).

q — Holds the matrix *Q* of size (*n*, *n*).

iwork — Holds the vector of length *n*.

jobu — Restored based on the presence of the argument *u* as follows:

jobu = 'U', if *u* is present, *jobu* = 'N', if *u* is omitted.

jobv — Restored based on the presence of the argument *v* as follows:

jobz = 'V', if *v* is present,

jobz = 'N', if *v* is omitted.

jobq — Restored based on the presence of the argument *q* as follows:

jobz = 'Q', if *q* is present,

jobz = 'N', if *q* is omitted.

?gesvdx
Computes the SVD and left and right singular vectors for a matrix.

Syntax

```
call sgesvdx(jobu, jobvt, range, m, n, a, lda, vl, vu, il, iu, ns, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
```

```
call dgesvdx(jobu, jobvt, range, m, n, a, lda, vl, vu, il, iu, ns, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
```

```
call cgesvdx(jobu, jobvt, range, m, n, a, lda, vl, vu, il, iu, ns, s, u, ldu, vt, ldvt,
work, lwork, rwork, iwork, info)
```

```
call zgesvdx(jobu, jobvt, range, m, n, a, lda, vl, vu, il, iu, ns, s, u, ldu, vt, ldvt,
work, lwork, rwork, iwork, info)
```

Include Files

- mkl.fi

Description

?gesvdx computes the singular value decomposition (SVD) of a real or complex m-by-n matrix *A*, optionally computing the left and right singular vectors. The SVD is written

A = *U* * *Σ* * transpose(*V*)

where *Σ* is an *m*-by-*n* matrix which is zero except for its min(*m*,*n*) diagonal elements, *U* is an *m*-by-*m* matrix, and *V* is an *n*-by-*n* matrix. The matrices *U* and *V* are orthogonal for real *A*, and unitary for complex *A*. The diagonal elements of *Σ* are the singular values of *A*; they are real and non-negative, and are returned in descending order. The first min(*m*,*n*) columns of *U* and *V* are the left and right singular vectors of *A*.

?gesvdx uses an eigenvalue problem for obtaining the SVD, which allows for the computation of a subset of singular values and vectors. See ?bdsvdx for details.

Note that the routine returns V^T, not *V*.

Input Parameters

jobu

CHARACTER*1. Specifies options for computing all or part of the matrix *U*:

= 'V': the first min(*m*,*n*) columns of *U* (the left singular vectors) or as specified by *range* are returned in the array *u*;

= 'N': no columns of *U* (no left singular vectors) are computed.

jobvt

CHARACTER*1. Specifies options for computing all or part of the matrix V^T:

= 'V': the first min(*m*,*n*) rows of V^T (the right singular vectors) or as specified by *range* are returned in the array *vt*;

= 'N': no rows of V^T (no right singular vectors) are computed.

range

CHARACTER*1. = 'A': find all singular values.

= 'V': all singular values in the half-open interval (*vl*,*vu*] are found.

= 'I': the *il*-th through *iu*-th singular values are found.

m

INTEGER. The number of rows of the input matrix *A*. $m \geq 0$.

n

INTEGER. The number of columns of the input matrix *A*. $n \geq 0$.

a

REAL for sgesvdx

DOUBLE PRECISION for dgesvdx

COMPLEX for cgesvdx

DOUBLE COMPLEX for zgesvdx

Array, size (*lda*,*n*)

On entry, the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.

lda≥ max(1,*m*).

vl REAL for sgesvdx

DOUBLE PRECISION for dgesvdx

REAL for cgesvdx

DOUBLE PRECISION for zgesvdx

vl≥0.

vu REAL for sgesvdx

DOUBLE PRECISION for dgesvdx

REAL for cgesvdx

DOUBLE PRECISION for zgesvdx

If *range*='V', the lower and upper bounds of the interval to be searched for singular values. *vu* > *vl*. Not referenced if *range* = 'A' or 'I'.

il INTEGER.

iu INTEGER. If *range*='I', the indices (in ascending order) of the smallest and largest singular values to be returned. 1 ≤*il*≤*iu*≤ min(*m*,*n*), if min(*m*,*n*) > 0. Not referenced if *range* = 'A' or 'V'.

ldu INTEGER. The leading dimension of the array *u*. *ldu*≥ 1; if *jobu* = 'V', *ldu*≥*m*.

ldvt INTEGER. The leading dimension of the array *vt*. *ldvt*≥ 1; if *jobvt* = 'V', *ldvt*≥*ns* (see above).

work REAL for sgesvdx

DOUBLE PRECISION for dgesvdx

COMPLEX for cgesvdx

DOUBLE COMPLEX for zgesvdx

Array, size (max(1,*lwork*)).

On exit, if info = 0, *work*(1) returns the optimal *lwork*;

lwork INTEGER. The size of the array *work*.

lwork≥ max(1, min(*m*, *n*)*(min(*m*, *n*) + 4)) for the paths (see comments inside the code):

- PATH 1 (*m* much larger than *n*)
- PATH 1t (*n* much larger than *m*)

lwork≥ max(1, min(*m*, *n*)*2 + max(*m*, *n*)) for the other paths. For good performance, *lwork* should generally be larger.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork

REAL for cgesvdx

DOUBLE PRECISION for zgesvdx

Array, size (max(1, *lrwork*)).

lrwork≥ min(*m*, *n*)*(min(*m*, *n*)*2 + 15*min(*m*, *n*)).

Output Parameters

a

On exit, the contents of *a* are destroyed.

ns

INTEGER. The total number of singular values found,

0 ≤*ns*≤ min(*m*, *n*).

If *range* = 'A', *ns* = min(*m*, *n*); if *range* = 'I', *ns* = *iu* - *il* + 1.

s

REAL for sgesvdx

DOUBLE PRECISION for dgesvdx

REAL for cgesvdx

DOUBLE PRECISION for zgesvdx

Array, size (min(*m*,*n*))

The singular values of *A*, sorted so that *s*(*i*)≥*s*(*i* + 1).

u

REAL for sgesvdx

DOUBLE PRECISION for dgesvdx

COMPLEX for cgesvdx

DOUBLE COMPLEX for zgesvdx

If *jobu* = 'V', *u* contains columns of *U* (the left singular vectors, stored columnwise) as specified by *range*; if *jobu* = 'N', *u* is not referenced.

NOTE
Make sure that *ucol*≥*ns*; if *range* = 'V', the exact value of *ns* is not known in advance and an upper bound must be used.

vt

REAL for sgesvdx

DOUBLE PRECISION for dgesvdx

COMPLEX for cgesvdx

DOUBLE COMPLEX for zgesvdx

Array, size (*ldvt*,*n*)

If *jobvt* = 'V', *vt* contains the rows of V^T (the right singular vectors, stored rowwise) as specified by *range*; if *jobvt* = 'N', *vt* is not referenced.

NOTE
Make sure that *ldvt*≥*ns*; if *range* = 'V', the exact value of *ns* is not known in advance and an upper bound must be used.

iwork INTEGER. Array, size (12*min(*m*, *n*)).

If *info* = 0, the first *ns* elements of *superb* are zero. If *info* > 0, then *superb* contains the indices of the eigenvectors that failed to converge in ?bdsvdx/?stevx.

info INTEGER.

= 0: successful exit.

< 0: if info = -*i*, the *i*-th argument had an illegal value.

> 0: if info = *i*, then *i* eigenvectors failed to converge in ?bdsvdx/?stevx. if info = *n**2 + 1, an internal error occurred in ?bdsvdx.

?bdsvdx
Computes the SVD of a bidiagonal matrix.

Syntax

```
call sbdsvdx (uplo, jobz, range, n, d, e, vl, vu, il, iu, ns, s, z, ldz, work, iwork, info )
```

```
call dbdsvdx (uplo, jobz, range, n, d, e, vl, vu, il, iu, ns, s, z, ldz, work, iwork, info )
```

Include Files

- mkl.fi

Description

?bdsvdx computes the singular value decomposition (SVD) of a real *n*-by-*n* (upper or lower) bidiagonal matrix *B*, *B* = *U* * *S* * *VT*, where *S* is a diagonal matrix with non-negative diagonal elements (the singular values of *B*), and *U* and *VT* are orthogonal matrices of left and right singular vectors, respectively.

Given an upper bidiagonal *B* with diagonal *d* = [$d_1 d_2$ ... d_n] and superdiagonal *e* = [$e_1 e_2$ ... e_{n-1}], ?bdsvdx computes the singular value decompositon of *B* through the eigenvalues and eigenvectors of the *n**2-by-*n**2 tridiagonal matrix

$$TGK = \begin{pmatrix} 0 & d_1 & & & \\ d_1 & 0 & e_1 & & \\ & e_1 & 0 & d_2 & \\ & & d_2 & \ddots & \ddots \\ & & & \ddots & \ddots \end{pmatrix}$$

If (*s*,*u*,*v*) is a singular triplet of *B* with ||*u*|| = ||*v*|| = 1, then (±*s*,*q*), ||*q*|| = 1, are eigenpairs of TGK, with $q = P * \frac{(u' \pm v')}{\sqrt{2}} = \frac{\begin{pmatrix} v_1 & u_1 & v_2 & u_2 & \cdots & v_n & u_n \end{pmatrix}}{\sqrt{2}}$, and $P = \begin{pmatrix} e_{n+1} & e_1 & e_{n+2} & e_2 & \cdots \end{pmatrix}$.

Given a TGK matrix, one can either

1. compute *-s*, *-v* and change signs so that the singular values (and corresponding vectors) are already in descending order (as in ?gesvd/?gesdd) or
2. compute *s*, *v* and reorder the values (and corresponding vectors).

?bdsvdx implements (1) by calling ?stevx (bisection plus inverse iteration, to be replaced with a version of the Multiple Relative Robust Representation algorithm. (See P. Willems and B. Lang, A framework for the MR^3 algorithm: theory and implementation, SIAM J. Sci. Comput., 35:740-766, 2013.)

Input Parameters

uplo

CHARACTER*1. = 'U': *B* is upper bidiagonal;

= 'L': B is lower bidiagonal.

jobz

CHARACTER*1. = 'N': Compute singular values only;

= 'V': Compute singular values and singular vectors.

range

CHARACTER*1. = 'A': Find all singular values.

= 'V': all singular values in the half-open interval [*vl*,*vu*) are found.

= 'I': the *il*-th through *iu*-th singular values are found.

n

INTEGER. The order of the bidiagonal matrix.

n >= 0.

d

REAL for sbdsvdx

DOUBLE PRECISION for dbdsvdx

Array, size *n*.

The *n* diagonal elements of the bidiagonal matrix *B*.

e

REAL for sbdsvdx

DOUBLE PRECISION for dbdsvdx

Array, size (max(1,*n* - 1))

The (*n* - 1) superdiagonal elements of the bidiagonal matrix *B* in elements 1 to *n* - 1.

vl

REAL for sbdsvdx

DOUBLE PRECISION for dbdsvdx

vl≥ 0.

vu

REAL for sbdsvdx

DOUBLE PRECISION for dbdsvdx

If *range*='V', the lower and upper bounds of the interval to be searched for singular values. *vu* > *vl*.

Not referenced if *range* = 'A' or 'I'.

il, *iu*

INTEGER. If *range*='I', the indices (in ascending order) of the smallest and largest singular values to be returned.

1 ≤*il*≤*iu*≤ min(*m*,*n*), if min(*m*,*n*) > 0.

Not referenced if *range* = 'A' or 'V'.

ldz INTEGER. The leading dimension of the array *z*.

ldz ≥ 1, and if *jobz* = 'V', *ldz* ≥ max(2,*n**2).

Output Parameters

ns INTEGER. The total number of singular values found. 0 ≤*ns*≤*n*.

If *range* = 'A', *ns* = *n*, and if *range* = 'I', *ns* = *iu* - *il* + 1.

s REAL for sbdsvdx

DOUBLE PRECISION for dbdsvdx

Array, size (*n*)

The first *ns* elements contain the selected singular values in ascending order.

z REAL for sbdsvdx

DOUBLE PRECISION for dbdsvdx

Array, size (2**n*, *k*)

If *jobz* = 'V', then if *info* = 0 the first *ns* columns of *z* contain the singular vectors of the matrix *B* corresponding to the selected singular values, with *U* in rows 1 to *n* and *V* in rows *n*+1 to *n**2, i.e.

$$z = \begin{pmatrix} U \\ V \end{pmatrix}$$

If *jobz* = 'N', then *z* is not referenced.

NOTE
Make sure that at least *k* = *ns*+1 columns are supplied in the array *z*; if *range* = 'V', the exact value of *ns* is not known in advance and an upper bound must be used.

work REAL for sbdsvdx

DOUBLE PRECISION for dbdsvdx

Array, size (14**n*)

iwork INTEGER. Array, size (12**n*).

If *jobz* = 'V', then if *info* = 0, the first *ns* elements of *iwork* are zero. If *info* > 0, then *iwork* contains the indices of the eigenvectors that failed to converge in ?stevx.

info INTEGER. = 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

> 0:

if *info* = *i*, then *i* eigenvectors failed to converge in ?stevx. The indices of the eigenvectors (as returned by ?stevx) are stored in the array *iwork*.

if *info* = *n**2 + 1, an internal error occurred.

Cosine-Sine Decomposition: LAPACK Driver Routines

This topic describes LAPACK driver routines for computing the *cosine-sine decomposition* (CS decomposition). You can also call the corresponding computational routines to perform the same task.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

Table "Driver Routines for Cosine-Sine Decomposition (CSD)" lists LAPACK routines (FORTRAN 77 interface) that perform CS decomposition of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Cosine-Sine Decomposition (CSD)

Operation	Real matrices	Complex matrices
Compute the CS decomposition of a block-partitioned orthogonal matrix	orcsd uncsd orcsd2by1 uncsd2by1	
Compute the CS decomposition of a block-partitioned unitary matrix		orcsd uncsd

See Also

CS Computational Routines

?orcsd/?uncsd
Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

Syntax

```
call sorcsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t, work,
lwork, iwork, info )

call dorcsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t, work,
lwork, iwork, info )

call cuncsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t, work,
lwork, rwork, lrwork, iwork, info )

call zuncsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t, work,
lwork, rwork, lrwork, iwork, info )

call orcsd( x11,x12,x21,x22,theta,u1,u2,v1t,v2t[,jobu1][,jobu2][,jobv1t][,jobv2t]
[,trans][,signs][,info] )

call uncsd( x11,x12,x21,x22,theta,u1,u2,v1t,v2t[,jobu1][,jobu2][,jobv1t][,jobv2t]
[,trans][,signs][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines `?orcsd`/`?uncsd` compute the CS decomposition of an *m*-by-*m* partitioned orthogonal matrix *X*:

$$X=\left(\begin{array}{c|c} x_{11} & x_{12} \\ \hline x_{21} & x_{22} \end{array}\right)=\left(\begin{array}{c|c} u_1 & \\ \hline & u_2 \end{array}\right)\left(\begin{array}{ccc|ccc} I & 0 & 0 & 0 & 0 \\ 0 & C & 0 & -S & 0 \\ 0 & 0 & 0 & 0 & -I \\ \hline 0 & 0 & 0 & I & 0 & 0 \\ 0 & S & 0 & 0 & C & 0 \\ 0 & 0 & I & 0 & 0 & 0 \end{array}\right)\left(\begin{array}{c|c} v_1 & \\ \hline & v_2 \end{array}\right)^T$$

or unitary matrix:

$$X=\left(\begin{array}{c|c} x_{11} & x_{12} \\ \hline x_{21} & x_{22} \end{array}\right)=\left(\begin{array}{c|c} u_1 & \\ \hline & u_2 \end{array}\right)\left(\begin{array}{ccc|ccc} I & 0 & 0 & 0 & 0 \\ 0 & C & 0 & -S & 0 \\ 0 & 0 & 0 & 0 & -I \\ \hline 0 & 0 & 0 & I & 0 & 0 \\ 0 & S & 0 & 0 & C & 0 \\ 0 & 0 & I & 0 & 0 & 0 \end{array}\right)\left(\begin{array}{c|c} v_1 & \\ \hline & v_2 \end{array}\right)^H$$

x_{11} is *p*-by-*q*. The orthogonal/unitary matrices u_1, u_2, v_1, and v_2 are *p*-by-*p*, *(m-p)*-by-*(m-p)*, *q*-by-*q*, *(m-q)*-by-*(m-q)*, respectively. *C* and *S* are *r*-by-*r* nonnegative diagonal matrices satisfying $C^2 + S^2 = I$, in which $r = \min(p, m-p, q, m-q)$.

Input Parameters

jobu1
: `CHARACTER`. If equals *Y*, then u_1 is computed. Otherwise, u_1 is not computed.

jobu2
: `CHARACTER`. If equals *Y*, then u_2 is computed. Otherwise, u_2 is not computed.

jobv1t
: `CHARACTER`. If equals *Y*, then v_1^t is computed. Otherwise, v_1^t is not computed.

jobv2t
: `CHARACTER`. If equals *Y*, then v_2^t is computed. Otherwise, v_2^t is not computed.

trans
: `CHARACTER`

  = 'T': *x*, u_1, u_2, v_1^t, v_2^t are stored in row-major order.

  otherwise: *x*, u_1, u_2, v_1^t, v_2^t are stored in column-major order.

signs
: `CHARACTER`

  = 'O': The lower-left block is made nonpositive (the "other" convention).

  otherwise: The upper-right block is made nonpositive (the "default" convention).

m	INTEGER. The number of rows and columns of the matrix *X*.
p	INTEGER. The number of rows in x_{11} and x_{12}. $0 \le p \le m$.
q	INTEGER. The number of columns in x_{11} and x_{21}. $0 \le q \le m$.
x11, *x12*, *x21*, *x22*	REAL for sorcsd DOUBLE PRECISION for dorcsd COMPLEX for cuncsd DOUBLE COMPLEX for zuncsd Arrays of size *x11* (*ldx11*,*q*), *x12* (*ldx12*,*m* - *q*), *x21* (*ldx21*,*q*), and *x22* (*ldx22*,*m* - *q*). Contain the parts of the orthogonal/unitary matrix whose CSD is desired.
ldx11, *ldx12*, *ldx21*, *ldx22*	INTEGER. The leading dimensions of the parts of array *X*. *ldx11* ≥ max(1, *p*), *ldx12* ≥ max(1, *p*), *ldx21* ≥ max(1, *m* - *p*), *ldx22* ≥ max(1, *m* - *p*).
ldu1	INTEGER. The leading dimension of the array u_1. If *jobu1* = '*Y*', *ldu1* ≥ max(1,*p*).
ldu2	INTEGER. The leading dimension of the array u_2. If *jobu2* = '*Y*', *ldu2* ≥ max(1,*m*-*p*).
ldv1t	INTEGER. The leading dimension of the array *v1t*. If *jobv1t* = '*Y*', *ldv1t* ≥ max(1,*q*).
ldv2t	INTEGER. The leading dimension of the array *v2t*. If *jobv2t* = '*Y*', *ldv2t* ≥ max(1,*m*-*q*).
work	REAL for sorcsd DOUBLE PRECISION for dorcsd COMPLEX for cuncsd DOUBLE COMPLEX for zuncsd Workspace array, size (max(1,*lwork*)).
lwork	INTEGER. The size of the *work* array. Constraints: If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
rwork	REAL for cuncsd DOUBLE PRECISION for zuncsd Workspace array, size (max(1,*lrwork*)).
lrwork	INTEGER. The size of the *rwork* array. Constraints:

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *rwork* array, returns this value as the first entry of the *rwork* array, and no error message related to *lrwork* is issued by xerbla.

iwork INTEGER. Workspace array, dimension *m*.

Output Parameters

theta REAL for sorcsd

DOUBLE PRECISION for dorcsd

COMPLEX for cuncsd

DOUBLE COMPLEX for zuncsd

Array, size (*r*), in which *r* = min(*p*,*m*-*p*,*q*,*m*-*q*).

C = diag(cos(*theta*(1)), ..., cos(*theta*(r))), and

S = diag(sin(*theta*(1)), ..., sin(*theta*(r))).

u1 REAL for sorcsd

DOUBLE PRECISION for dorcsd

COMPLEX for cuncsd

DOUBLE COMPLEX for zuncsd

Array, size (*p*).

If *jobu1* = 'Y', *u1* contains the *p*-by-*p* orthogonal/unitary matrix u_1.

u2 REAL for sorcsd

DOUBLE PRECISION for dorcsd

COMPLEX for cuncsd

DOUBLE COMPLEX for zuncsd

Array, size (*ldu2*,*m*-*p*).

If *jobu2* = 'Y', *u2* contains the (*m*-*p*)-by-(*m*-*p*) orthogonal/unitary matrix u_2.

v1t REAL for sorcsd

DOUBLE PRECISION for dorcsd

COMPLEX for cuncsd

DOUBLE COMPLEX for zuncsd

Array, size (*ldv1t*,*) .

If *jobv1t* = 'Y', *v1t* contains the *q*-by-*q* orthogonal matrix v_1^T or unitary matrix v_1^H.

v2t REAL for sorcsd

DOUBLE PRECISION for dorcsd

COMPLEX for cuncsd

DOUBLE COMPLEX for zuncsd

Array, size (*ldv2t*,*m*-*q*).

If *jobv2t* = 'Y', *v2t* contains the (*m*-*q*)-by-(*m*-*q*) orthogonal matrix v_2^T or unitary matrix v_2^H.

work On exit,

If *info* = 0, *work(1)* returns the optimal *lwork*.

If *info* > 0, For ?orcsd, *work(2:r)* contains the values *phi(1), ..., phi(r-1)* that, together with *theta(1), ..., theta(r)* define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. *info* specifies the number of nonzero *phi*'s.

rwork On exit,

If *info* = 0, *rwork(1)* returns the optimal *lrwork*.

If *info* > 0, For ?uncsd, *rwork(2:r)* contains the values *phi(1), ..., phi(r-1)* that, together with *theta(1), ..., theta(r)* define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. *info* specifies the number of nonzero *phi*'s.

info INTEGER.

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument has an illegal value

> 0: ?orcsd/?uncsd did not converge. See the description of *work* above for details.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran 95 Interface Conventions.

Specific details for the routine ?orcsd/?uncsd interface are as follows:

x11 Holds the block of matrix *X* of size (*p*, *q*).

x12 Holds the block of matrix *X* of size (*p*, *m*-*q*).

x21 Holds the block of matrix *X* of size (*m*-*p*, *q*).

x22 Holds the block of matrix *X* of size (*m*-*p*, *m*-*q*).

theta Holds the vector of length *r* = min(*p*,*m*-*p*,*q*,*m*-*q*).

u1 Holds the matrix of size (*p*,*p*).

u2 Holds the matrix of size (*m*-*p*,*m*-*p*).

v1t Holds the matrix of size (*q*,*q*).

v2t	Holds the matrix of size (*m-q*,*m-q*).
jobsu1	Indicates whether u_1 is computed. Must be 'Y' or 'O'.
jobsu2	Indicates whether u_2 is computed. Must be 'Y' or 'O'.
jobv1t	Indicates whether v_1^t is computed. Must be 'Y' or 'O'.
jobv2t	Indicates whether v_2^t is computed. Must be 'Y' or 'O'.
trans	Must be 'N' or 'T'.
signs	Must be 'O' or 'D'.

See Also

?bbcsd

xerbla

?orcsd2by1/?uncsd2by1
Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

Syntax

```
call sorcsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1, ldu1,
u2, ldu2, v1t, ldv1t, work, lwork, iwork, info )
```

```
call dorcsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1, ldu1,
u2, ldu2, v1t, ldv1t, work, lwork, iwork, info )
```

```
call cuncsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1, ldu1,
u2, ldu2, v1t, ldv1t, work, lwork, rwork, lrwork, iwork, info )
```

```
call zuncsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1, ldu1,
u2, ldu2, v1t, ldv1t, work, lwork, rwork, lrwork, iwork, info )
```

```
call orcsd2by1( x11,x21,theta,u1,u2,v1t[,jobu1][,jobu2][,jobv1t][,info] )
```

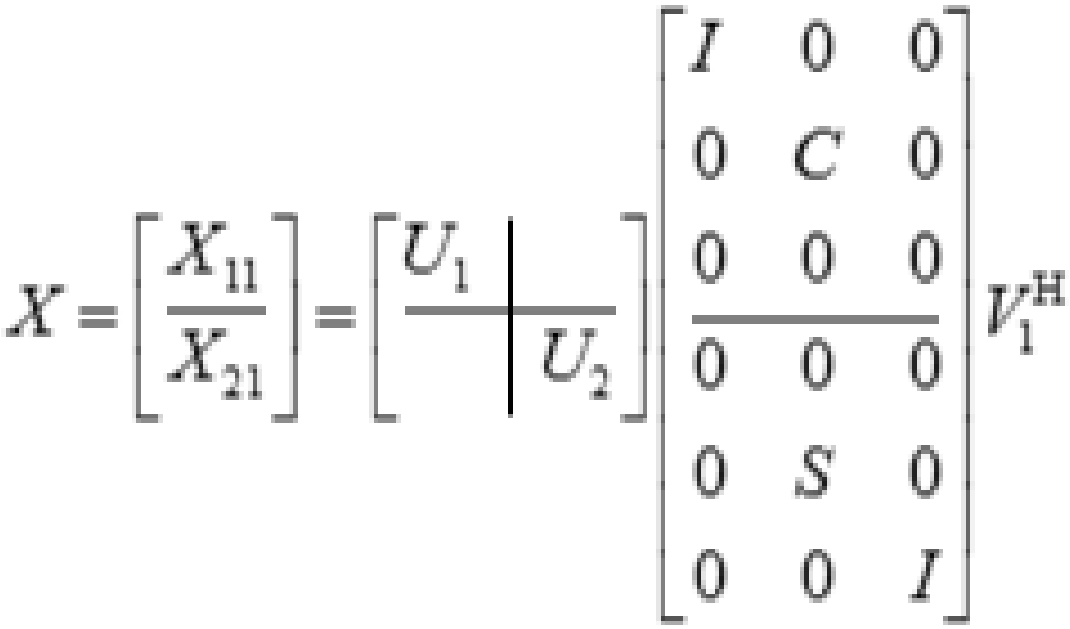
```
call uncsd2by1( x11,x21,theta,u1,u2,v1t[,jobu1][,jobu2][,jobv1t][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines ?orcsd2by1/?uncsd2by1 compute the CS decomposition of an *m*-by-*q* matrix *X* with orthonormal columns that has been partitioned into a 2-by-1 block structure:



x_{11} is p-by-q. The orthogonal/unitary matrices u_1, u_2, v_1, and v_2 are p-by-p, $(m-p)$-by-$(m-p)$, q-by-q, $(m-q)$-by-$(m-q)$, respectively. C and S are r-by-r nonnegative diagonal matrices satisfying $C^2 + S^2 = I$, in which $r = \min(p, m-p, q, m-q)$.

Input Parameters

jobu1	CHARACTER. If equal to 'Y', then u_1 is computed. Otherwise, u_1 is not computed.
jobu2	CHARACTER. If equal to 'Y', then u_2 is computed. Otherwise, u_2 is not computed.
jobv1t	CHARACTER. If equal to 'Y', then v_1^t is computed. Otherwise, v_1^t is not computed.
m	INTEGER. The number of rows and columns of the matrix X.
p	INTEGER. The number of rows in x_{11}. $0 \le p \le m$.
q	INTEGER. The number of columns in x_{11} . $0 \le q \le m$.
x11	REAL for sorcsd2by1 DOUBLE PRECISION for dorcsd2by1 COMPLEX for cuncsd2by1 DOUBLE COMPLEX for zuncsd2by1 Array, size (*ldx11*,*q*). On entry, the part of the orthogonal matrix whose CSD is desired.
ldx11	INTEGER. The leading dimension of the array *x11*. *ldx11* $\ge \max(1, p)$.
x21	REAL for sorcsd2by1 DOUBLE PRECISION for dorcsd2by1

COMPLEX for cuncsd2by1

DOUBLE COMPLEX for zuncsd2by1

Array, size (*ldx21*,*q*).

On entry, the part of the orthogonal matrix whose CSD is desired.

ldx21 INTEGER. The leading dimension of the array *X*. *ldx21* ≥ max(1,*m* - *p*).

ldu1 INTEGER. The leading dimension of the array u_1. If *jobu1* = '*Y*', *ldu1* ≥ max(1,*p*).

ldu2 INTEGER. The leading dimension of the array u_2. If *jobu2* = '*Y*', *ldu2* ≥ max(1,*m*-*p*).

ldv1t INTEGER. The leading dimension of the array *v1t*. If *jobv1t* = '*Y*', *ldv1t* ≥ max(1,*q*).

work REAL for sorcsd2by1

DOUBLE PRECISION for dorcsd2by1

COMPLEX for cuncsd2by1

DOUBLE COMPLEX for zuncsd2by1

Workspace array, size (max(1,*lwork*)).

lwork INTEGER. The size of the *work* array. Constraints:

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork REAL for cuncsd2by1

DOUBLE PRECISION for zuncsd2by1

Workspace array, size (max(1,*lrwork*)).

lrwork INTEGER. The size of the *rwork* array. Constraints:

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *rwork* array, returns this value as the first entry of the *rwork* array, and no error message related to *lrwork* is issued by xerbla.

iwork INTEGER. Workspace array, dimension *m* - min(*p*, *m* - *p*, *q*, *m* - *q*).

Output Parameters

theta REAL for sorcsd2by1

DOUBLE PRECISION for dorcsd2by1

COMPLEX for cuncsd2by1

DOUBLE COMPLEX for zuncsd2by1

Array, size (*r*), in which *r* = min(*p*,*m*-*p*,*q*,*m*-*q*).

C = diag(cos(*theta*(1)), ..., cos(*theta*(r))), and

S = diag(sin(*theta*(1)), ..., sin(*theta*(r))).

u1

REAL for sorcsd2by1

DOUBLE PRECISION for dorcsd2by1

COMPLEX for cuncsd2by1

DOUBLE COMPLEX for zuncsd2by1

Array, size (*ldu1*,*p*) .

If *jobu1* = '*Y*', *u1* contains the *p*-by-*p* orthogonal/unitary matrix u_1.

u2

REAL for sorcsd2by1

DOUBLE PRECISION for dorcsd2by1

COMPLEX for cuncsd2by1

DOUBLE COMPLEX for zuncsd2by1

Array, size (*ldu2*,*m* - *p*) .

If *jobu2* = '*Y*', *u2* contains the (*m*-*p*)-by-(*m*-*p*) orthogonal/unitary matrix u_2.

v1t

REAL for sorcsd2by1

DOUBLE PRECISION for dorcsd2by1

COMPLEX for cuncsd2by1

DOUBLE COMPLEX for zuncsd2by1

Array, size (*ldv1t*,*q*) .

If *jobv1t* = '*Y*', *v1t* contains the *q*-by-*q* orthogonal matrix v_1^T or unitary matrix v_1^H.

work

On exit,

If *info* = 0, *work(1)* returns the optimal *lwork*.

If *info* > 0, *work(2:r)* contains the values *phi(1)*, ..., *phi(r-1)* that, together with *theta(1)*, ..., *theta(r)* define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. *info* specifies the number of nonzero *phi*'s.

rwork

On exit,

If *info* = 0, *rwork(1)* returns the optimal *lrwork*.

If *info* > 0, *rwork(2:r)* contains the values *phi(1)*, ..., *phi(r-1)* that, together with *theta(1)*, ..., *theta(r)* define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. *info* specifies the number of nonzero *phi*'s.

info

INTEGER.

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument has an illegal value

> 0: ?orcsd2by1/?uncsd2by1 did not converge. See the description of *work* above for details.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran 95 Interface Conventions.

Specific details for the routine ?orcsd2by1/?orcsd2by1 interface are as follows:

x11 Holds the block of matrix X_{11} of size (p, q).

x21 Holds the block of matrix X_{21} of size (m-p, q).

theta Holds the vector of length $r = \min(p, m-p, q, m-q)$.

u1 Holds the matrix u_1 of size (p,p).

u2 Holds the matrix u_2 of size (m-p,m-p).

v1t Holds the matrix v_1^T or v_1^H of size (q,q).

jobu1 Indicates whether u_1 is computed. Must be 'Y' or 'O'.

jobu2 Indicates whether u_2 is computed. Must be 'Y' or 'O'.

jobv1t Indicates whether v_1^t is computed. Must be 'Y' or 'O'.

See Also

?bbcsd

xerbla

Generalized Symmetric Definite Eigenvalue Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving generalized symmetric definite eigenproblems. See also computational routines that can be called to solve these problems. Table "Driver Routines for Solving Generalized Symmetric Definite Eigenproblems" lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Generalized Symmetric Definite Eigenproblems

Routine Name	Operation performed
sygv/hegv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.
sygvd/hegvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.
sygvx/hegvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.
spgv/hpgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.

Routine Name	Operation performed
spgvd/hpgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.
spgvx/hpgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.
sbgv/hbgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.
sbgvd/hbgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.
sbgvx/hbgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.

?sygv
Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
call ssygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
call dsygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
call sygv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$A*x = \lambda*B*x$, $A*B*x = \lambda*x$, or $B*A*x = \lambda*x$.

Here *A* and *B* are assumed to be symmetric and *B* is also positive definite.

Input Parameters

itype
INTEGER. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is *A*x = lambda*B*x*;
if *itype* = 2, the problem type is *A*B*x = lambda*x*;
if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz
CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo — CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

a, *b*, *work* — REAL for ssygv

DOUBLE PRECISION for dsygv.

Arrays:

a(*lda*,*) contains the upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the upper or lower triangle of the symmetric positive definite matrix *B*, as specified by *uplo*.

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The leading dimension of *b*; at least max(1, *n*).

lwork — INTEGER.

The dimension of the array *work*;

lwork≥ max(1, 3*n*-1).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a

On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^T*B*Z = I$;

if *itype* = 3, $Z^T*\text{inv}(B)*Z = I$;

If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b

On exit, if *info*≤*n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$.

w — REAL for ssygv

DOUBLE PRECISION for dsygv.

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, spotrf/dpotrf or ssyev/dsyev returned an error code:

If *info* = *i*≤*n*, ssyev/dsyev failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sygv interface are the following:

a — Holds the matrix *A* of size (*n*, *n*).

b — Holds the matrix *B* of size (*n*, *n*).

w — Holds the vector of length *n*.

itype — Must be 1, 2, or 3. The default value is 1.

jobz — Must be 'N' or 'V'. The default value is 'N'.

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use *lwork*≥ (*nb*+2)*n, where *nb* is the blocksize for ssytrd/dsytrd returned by ilaenv.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *work* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegv
Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

```
call chegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
call zhegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
call hegv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

*A*x* = λ**B*x*, *A*B*x* = λ**x*, or *B*A*x* = λ**x*.

Here *A* and *B* are assumed to be Hermitian and *B* is also positive definite.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is *A*x = lambda*B*x*;

if *itype* = 2, the problem type is *A*B*x = lambda*x*;

if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

a, *b*, *work*

COMPLEX for chegv

DOUBLE COMPLEX for zhegv.

Arrays:

a(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*.

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

lda — `INTEGER`. The leading dimension of *a*; at least max(1, *n*).

ldb — `INTEGER`. The leading dimension of *b*; at least max(1, *n*).

lwork — `INTEGER`.

The dimension of the array *work*; $lwork \geq \max(1, 2n-1)$.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

rwork — `REAL` for `chegv`

`DOUBLE PRECISION` for `zhegv`.

Workspace array, size at least max(1, 3*n*-2).

Output Parameters

a — On exit, if `jobz = 'V'`, then if `info = 0`, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if `itype = 1` or 2, $Z^H*B*Z = I$;

if `itype = 3`, $Z^H*\text{inv}(B)*Z = I$;

If `jobz = 'N'`, then on exit the upper triangle (if `uplo = 'U'`) or the lower triangle (if `uplo = 'L'`) of *A*, including the diagonal, is destroyed.

b — On exit, if $info \leq n$, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$.

w — `REAL` for `chegv`

`DOUBLE PRECISION` for `zhegv`.

Array, size at least max(1, *n*).

If `info = 0`, contains the eigenvalues in ascending order.

work(1) — On exit, if `info = 0`, then *work(1)* returns the required minimal size of *lwork*.

info — `INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th argument has an illegal value.

If `info > 0`, `cpotrf`/`zpotrf` or `cheev`/`zheev` return an error code:

If $info = i \leq n$, `cheev`/`zheev` fails to converge, and *i* off-diagonal elements of an intermediate tridiagonal do not converge to zero;

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* can not be completed and no eigenvalues or eigenvectors are computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hegv` interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length *n*.
itype	Must be 1, 2, or 3. The default value is 1.
jobz	Must be `'N'` or `'V'`. The default value is `'N'`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

For optimum performance use *lwork*≥ (*nb*+1)**n*, where *nb* is the blocksize for `chetrd`/`zhetrd` returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

Syntax

```
call ssygvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, iwork, liwork, info)
call dsygvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, iwork, liwork, info)
call sygvd(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

*A*x = λ*B*x*, *A*B*x = λ*x*, or *B*A*x = λ*x*.

Here *A* and *B* are assumed to be symmetric and *B* is also positive definite.

It uses a divide and conquer algorithm.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is *A*x = lambda*B*x*;

if *itype* = 2, the problem type is *A*B*x = lambda*x*;

if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

a, *b*, *work*

REAL for ssygvd

DOUBLE PRECISION for dsygvd.

Arrays:

a(*lda*,*) contains the upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the upper or lower triangle of the symmetric positive definite matrix *B*, as specified by *uplo*.

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldb

INTEGER. The leading dimension of *b*; at least max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

If *n*≤ 1, *lwork*≥ 1;

If *jobz* = 'N' and *n*>1, *lwork* < 2*n*+1;

If *jobz* = 'V' and *n*>1, *lwork* < $2n^2+6n+1$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork

INTEGER.

Workspace array, its dimension max(1, *lwork*).

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

If *n*≤ 1, *liwork*≥ 1;

If *jobz* = 'N' and *n*>1, *liwork*≥ 1;

If *jobz* = 'V' and *n*>1, *liwork*≥ 5*n*+3.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

a

On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^T*B*Z = I$;

if *itype* = 3, $Z^T*\text{inv}(B)*Z = I$;

If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b

On exit, if *info*≤*n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$.

w

REAL for ssygvd

DOUBLE PRECISION for dsygvd.

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, an error code is returned as specified below.

- For *info*≤*n*:
  - If *info* = *i* and *jobz* = 'N', then the algorithm failed to converge; *i* off-diagonal elements of an intermediate tridiagonal form did not converge to zero.
  - If *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n* +1) through mod(*info*,*n*+1).
- For *info* > *n*:
  - If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sygvd interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length *n*.
itype	Must be 1, 2, or 3. The default value is 1.
jobz	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *work* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegvd
Computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem using a divide and conquer method.

Syntax

call chegvd(*itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call zhegvd(*itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call hegvd(*a*, *b*, *w* [,*itype*] [,*jobz*] [,*uplo*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

*A*x* = λ*B*x*, *A*B*x* = λ*x*, or *B*A*x* = λ*x*.

Here *A* and *B* are assumed to be Hermitian and *B* is also positive definite.

It uses a divide and conquer algorithm.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is *A*x = lambda*B*x*;

if *itype* = 2, the problem type is *A*B*x = lambda*x*;

if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

a, *b*, *work*

COMPLEX for chegvd

DOUBLE COMPLEX for zhegvd.

Arrays:

a(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*.

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

lda `INTEGER`. The leading dimension of *a*; at least max(1, *n*).

ldb `INTEGER`. The leading dimension of *b*; at least max(1, *n*).

lwork `INTEGER`.

The dimension of the array *work*.

Constraints:

If $n \leq 1$, $lwork \geq 1$;

If *jobz* = 'N' and $n>1$, $lwork \geq n+1$;

If *jobz* = 'V' and $n>1$, $lwork \geq n^2+2n$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

rwork `REAL` for `chegvd`

`DOUBLE PRECISION` for `zhegvd`.

Workspace array, size `max(1, lrwork)`.

lrwork `INTEGER`.

The dimension of the array *rwork*.

Constraints:

If $n \leq 1$, $lrwork \geq 1$;

If *jobz* = 'N' and $n>1$, $lrwork \geq n$;

If *jobz* = 'V' and $n>1$, $lrwork \geq 2n^2+5n+1$.

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork `INTEGER`.

Workspace array, size `max(1, liwork)`.

liwork `INTEGER`.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $liwork \geq 1$;

If *jobz* = 'N' and $n>1$, $liwork \geq 1$;

If *jobz* = 'V' and $n>1$, $liwork \geq 5n+3$.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

a

On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^H * B * Z = I$;

if *itype* = 3, $Z^H * \mathrm{inv}(B) * Z = I$;

If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b

On exit, if *info*≤*n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.

w

REAL for chegvd

DOUBLE PRECISION for zhegvd.

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork(1)

On exit, if *info* = 0, then *rwork*(1) returns the required minimal size of *lrwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* = *i*, and *jobz* = 'N', then the algorithm failed to converge; *i* off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

if *info* = *i*, and *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*, *n*+1).

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hegvd` interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length *n*.
itype	Must be 1, 2, or 3. The default value is 1.
jobz	Must be `'N'` or `'V'`. The default value is `'N'`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
call ssygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
```

```
call dsygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
```

```
call sygvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

*A*x* = λ**B*x*, *A*B*x* = λ**x*, or *B*A*x* = λ**x*.

Here *A* and *B* are assumed to be symmetric and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is *A*x* = λ**B*x*;

if *itype* = 2, the problem type is *A*B*x* = λ**x*;

if *itype* = 3, the problem type is *B*A*x* = λ**x*.

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range

CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *lambda*(i) in the half-open interval:

vl<*lambda*(i)≤*vu*.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

a, *b*, *work*

REAL for ssygvx

DOUBLE PRECISION for dsygvx.

Arrays:

a(*lda*,*) contains the upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the upper or lower triangle of the symmetric positive definite matrix *B*, as specified by *uplo*.

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldb

INTEGER. The leading dimension of *b*; at least max(1, *n*).

vl, *vu*

REAL for ssygvx

DOUBLE PRECISION for dsygvx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl*< *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0

if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for ssygvx

DOUBLE PRECISION for dsygvx. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array *z*. Constraints:

ldz≥ 1; if *jobz* = 'V', *ldz*≥ max(1, *n*) .

lwork

INTEGER.

The dimension of the array *work*;

lwork < max(1, 8*n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

iwork

INTEGER.

Workspace array, size at least max(1, 5*n*).

Output Parameters

a

On exit, the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is overwritten.

b

On exit, if *info*≤*n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$.

m

INTEGER. The total number of eigenvalues found,

0 ≤*m*≤*n*. If *range* = 'A', *m* = *n*, and if *range* = 'I',

m = *iu*-*il*+1.

w, *z*

REAL for ssygvx

DOUBLE PRECISION for dsygvx.

Arrays:

w(*), size at least max(1, *n*).

The first *m* elements of *w* contain the selected eigenvalues in ascending order.

`z`(*ldz*,*) .

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^T*B*Z = I$;

if *itype* = 3, $Z^T*\mathrm{inv}(B)*Z = I$;

If *jobz* = 'N', then *z* is not referenced.

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

work(1)

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

ifail

INTEGER.

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th argument had an illegal value.

If *info* > 0, `spotrf`/`dpotrf` and `ssyevx`/`dsyevx` returned an error code:

If *info* = *i*≤*n*, `ssyevx`/`dsyevx` failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `sygvx` interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length *n*.
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length *n*.
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted. Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*$||T||_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *C* to tridiagonal form, where *C* is the symmetric matrix of the standard symmetric problem to which the generalized problem is transformed. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, set *abstol* to 2*?lamch('S').

For optimum performance use *lwork*≥ (*nb*+3)*n, where *nb* is the blocksize for ssytrd/dsytrd returned by ilaenv.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegvx
Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

call chegvx(*itype*, *jobz*, *range*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *iwork*, *ifail*, *info*)

call zhegvx(*itype*, *jobz*, *range*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *iwork*, *ifail*, *info*)

call hegvx(*a*, *b*, *w* [,*itype*] [,*uplo*] [,*z*] [,*vl*] [,*vu*] [,*il*] [,*iu*] [,*m*] [,*ifail*] [,*abstol*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

*A*x* = λ*B*x*, *A*B*x* = λ*x*, or *B*A*x* = λ*x*.

Here *A* and *B* are assumed to be Hermitian and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is *A*x* = λ*B*x*;

if *itype* = 2, the problem type is *A*B*x* = λ*x*;

if *itype* = 3, the problem type is *B*A*x* = λ*x*.

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range

CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *lambda*(i) in the half-open interval:

	vl<*lambda*(i)≤*vu*.
	If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
uplo	CHARACTER*1. Must be 'U' or 'L'.
	If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*;
	If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).
a, *b*, *work*	COMPLEX for chegvx
	DOUBLE COMPLEX for zhegvx.
	Arrays:
	a(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
	The second dimension of *a* must be at least max(1, *n*).
	b(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*.
	The second dimension of *b* must be at least max(1, *n*).
	work is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; at least max(1, *n*).
vl, *vu*	REAL for chegvx
	DOUBLE PRECISION for zhegvx.
	If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
	Constraint: *vl*< *vu*.
	If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER.
	If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
	Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0
	if *n* = 0.
	If *range* = 'A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for chegvx
	DOUBLE PRECISION for zhegvx.
	The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.
ldz	INTEGER. The leading dimension of the output array *z*. Constraints:
	ldz≥ 1; if *jobz* = 'V', *ldz*≥ max(1, *n*).

lwork INTEGER.

The dimension of the array *work*; *lwork* ≥ max(1, 2*n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See *Application Notes* for the suggested value of *lwork*.

rwork REAL for chegvx

DOUBLE PRECISION for zhegvx.

Workspace array, size at least max(1, 7*n*).

iwork INTEGER.

Workspace array, size at least max(1, 5*n*).

Output Parameters

a On exit, the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is overwritten.

b On exit, if *info*≤*n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$.

m INTEGER. The total number of eigenvalues found,

0 ≤*m*≤*n*. If *range* = 'A', *m* = *n*, and if *range* = 'I',

m = *iu*-*il*+1.

w REAL for chegvx

DOUBLE PRECISION for zhegvx.

Array, size at least max(1, *n*).

The first *m* elements of *w* contain the selected eigenvalues in ascending order.

z COMPLEX for chegvx

DOUBLE COMPLEX for zhegvx.

Array z(*ldz*,*). The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows:

if *itype* = 1 or 2, Z^H*B*Z = I;

if *itype* = 3, Z^H*inv(*B*)*Z* = I;

If *jobz* = 'N', then *z* is not referenced.

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if `range = 'V'`, the exact value of *m* is not known in advance and an upper bound must be used.

work(1) On exit, if `info = 0`, then *work(1)* returns the required minimal size of *lwork*.

ifail `INTEGER`.

Array, size at least max(1, *n*).

If `jobz = 'V'`, then if `info = 0`, the first *m* elements of *ifail* are zero; if `info > 0`, the *ifail* contains the indices of the eigenvectors that failed to converge.

If `jobz = 'N'`, then *ifail* is not referenced.

info `INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*th argument had an illegal value.

If `info > 0`, `cpotrf`/`zpotrf` and `cheevx`/`zheevx` returned an error code:

If `info = i≤n`, `cheevx`/`zheevx` failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If `info = n + i`, for `1 ≤i≤n`, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hegvx` interface are the following:

a Holds the matrix *A* of size (*n*, *n*).

b Holds the matrix *B* of size (*n*, *n*).

w Holds the vector of length *n*.

z Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.

ifail Holds the vector of length *n*.

itype Must be 1, 2, or 3. The default value is 1.

uplo Must be `'U'` or `'L'`. The default value is `'U'`.

vl Default value for this element is *vl* = `-HUGE`(*vl*).

vu Default value for this element is *vu* = `HUGE`(*vl*).

il Default value for this argument is `il = 1`.

iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted. Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*$||T||_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *C* to tridiagonal form, where *C* is the symmetric matrix of the standard symmetric problem to which the generalized problem is transformed. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

For optimum performance use *lwork*≥ (*nb*+1)**n*, where *nb* is the blocksize for chetrd/zhetrd returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgv
Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

```
call sspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
call dspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
```

```
call spgv(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

*A*x* = λ**B*x*, *A*B*x* = λ**x*, or *B*A*x* = λ**x*.

Here *A* and *B* are assumed to be symmetric, stored in packed format, and *B* is also positive definite.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x = lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).
ap, *bp*, *work*	REAL for sspgv DOUBLE PRECISION for dspgv. Arrays: *ap*(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, *n**(*n*+1)/2). *bp*(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*. The dimension of *bp* must be at least max(1, *n**(*n*+1)/2). *work(*)* is a workspace array, size at least max(1, 3*n*).
ldz	INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1. If *jobz* = 'V', *ldz*≥ max(1, *n*).

Output Parameters

ap	On exit, the contents of *ap* are overwritten.

bp — On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$, in the same storage format as *B*.

w, *z* — REAL for sspgv

DOUBLE PRECISION for dspgv.

Arrays:

w(*), size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*) .

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^T*B*Z = I$;

if *itype* = 3, $Z^T*\text{inv}(B)*Z = I$;

If *jobz* = 'N', then *z* is not referenced.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, spptrf/dpptrf and sspev/dspev returned an error code:

If *info* = *i*≤*n*, sspev/dspev failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine spgv interface are the following:

ap — Holds the array *A* of size (*n**(*n*+1)/2).

bp — Holds the array *B* of size (*n**(*n*+1)/2).

w — Holds the vector with the number of elements *n*.

z — Holds the matrix *Z* of size (*n*, *n*).

itype — Must be 1, 2, or 3. The default value is 1.

uplo — Must be 'U' or 'L'. The default value is 'U'.

jobz — Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

?hpgv
Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with matrices in packed storage.

Syntax

```
call chpgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
call zhpgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
call hpgv(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$A*x = \lambda*B*x$, $A*B*x = \lambda*x$, or $B*A*x = \lambda*x$.

Here *A* and *B* are assumed to be Hermitian, stored in packed format, and *B* is also positive definite.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is *A*x = lambda*B*x*;

if *itype* = 2, the problem type is *A*B*x = lambda*x*;

if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, *bp*, *work*

COMPLEX for chpgv

DOUBLE COMPLEX for zhpgv.

Arrays:

ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.

The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*.

The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).

work(*) is a workspace array, size at least max(1, 2*n*-1).

ldz — INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).

rwork — REAL for chpgv

DOUBLE PRECISION for zhpgv.

Workspace array, size at least max(1, 3*n*-2).

Output Parameters

ap — On exit, the contents of *ap* are overwritten.

bp — On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$, in the same storage format as *B*.

w — REAL for chpgv

DOUBLE PRECISION for zhpgv.

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z — COMPLEX for chpgv

DOUBLE COMPLEX for zhpgv.

Array z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^H*B*Z = I$;

if *itype* = 3, $Z^H*\text{inv}(B)*Z = I$;

If *jobz* = 'N', then *z* is not referenced.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, cpptrf/zpptrf and chpev/zhpev returned an error code:

If *info* = *i*≤*n*, chpev/zhpev failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hpgv` interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
bp	Holds the array *B* of size (*n*(n+1)/2*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted.

?spgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage using a divide and conquer method.

Syntax

```
call sspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork, liwork, info)
call dspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork, liwork, info)
call spgvd(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

*A*x* = λ**B*x*, *A*B*x* = λ**x*, or *B*A*x* = λ**x*.

Here *A* and *B* are assumed to be symmetric, stored in packed format, and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype	`INTEGER`. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x = lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	`CHARACTER*1`. Must be `'N'` or `'V'`. If *jobz* = `'N'`, then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, *bp*, *work*

REAL for sspgvd

DOUBLE PRECISION for dspgvd.

Arrays:

ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.

The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*.

The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).

work is a workspace array, its dimension max(1, *lwork*).

ldz

INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

If $n \leq 1$, $lwork \geq 1$;

If *jobz* = 'N' and $n > 1$, $lwork \geq 2n$;

If *jobz* = 'V' and $n > 1$, $lwork \geq 2n^2+6n+1$.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork

INTEGER.

Workspace array, dimension max(1, *lwork*).

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $liwork \geq 1$;

If *jobz* = 'N' and $n > 1$, $liwork \geq 1$;

If *jobz* = 'V' and $n > 1$, $liwork \geq 5n+3$.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

ap

On exit, the contents of *ap* are overwritten.

bp

On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$, in the same storage format as *B*.

w, *z*

REAL for sspgv

DOUBLE PRECISION for dspgv.

Arrays:

w(*), size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^T*B*Z = I$;

if *itype* = 3, $Z^T*\mathrm{inv}(B)*Z = I$;

If *jobz* = 'N', then *z* is not referenced.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, spptrf/dpptrf and sspevd/dspevd returned an error code:

If *info* = *i*≤*n*, sspevd/dspevd failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine spgvd interface are the following:

ap	Holds the array *A* of size (*n*(n+1)/2*).
bp	Holds the array *B* of size (*n*(n+1)/2*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hpgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with matrices in packed storage using a divide and conquer method.

Syntax

```
call chpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zhpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call hpgvd(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

*A*x* = λ**B*x*, *A*B*x* = λ**x*, or *B*A*x* = λ**x*.

Here *A* and *B* are assumed to be Hermitian, stored in packed format, and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype

INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is *A*x = lambda*B*x*;

if *itype* = 2, the problem type is *A*B*x = lambda*x*;

if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

ap, *bp*, *work*

COMPLEX for chpgvd

DOUBLE COMPLEX for zhpgvd.

Arrays:

ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.

The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*.

The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).

work is a workspace array, its dimension max(1, *lwork*).

ldz

INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1. If *jobz* = 'V', *ldz*≥ max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

If *n*≤ 1, *lwork*≥ 1;

If *jobz* = 'N' and *n*>1, *lwork*≥*n*;

If *jobz* = 'V' and *n*>1, *lwork*≥ 2*n*.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

rwork

REAL for chpgvd

DOUBLE PRECISION for zhpgvd.

Workspace array, its dimension max(1, *lrwork*).

lrwork

INTEGER.

The dimension of the array *rwork*.

Constraints:

If $n \leq 1$, $lrwork \geq 1$;

If *jobz* = 'N' and $n>1$, $lrwork \geq n$;

If *jobz* = 'V' and $n>1$, $lrwork \geq 2n^2+5n+1$.

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork

INTEGER.

Workspace array, its dimension max(1, *liwork*).

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $liwork \geq 1$;

If *jobz* = 'N' and $n>1$, $liwork \geq 1$;

If *jobz* = 'V' and $n>1$, $liwork \geq 5n+3$.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

ap

On exit, the contents of *ap* are overwritten.

bp

On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$, in the same storage format as *B*.

w

REAL for chpgvd

DOUBLE PRECISION for zhpgvd.

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z

COMPLEX for chpgvd

DOUBLE COMPLEX for zhpgvd.

Array z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^H*B*Z = I$;

if *itype* = 3, $Z^H*\mathrm{inv}(B)*Z = I$;

If *jobz* = 'N', then *z* is not referenced.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork(1)

On exit, if *info* = 0, then *rwork*(1) returns the required minimal size of *lrwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, cpptrf/zpptrf and chpevd/zhpevd returned an error code:

If *info* = *i*≤*n*, chpevd/zhpevd failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hpgvd interface are the following:

ap

Holds the array *A* of size (*n**(*n*+1)/2).

bp

Holds the array *B* of size (*n**(*n*+1)/2).

w

Holds the vector with the number of elements *n*.

z

Holds the matrix *Z* of size (*n*, *n*).

itype

Must be 1, 2, or 3. The default value is 1.

uplo

Must be 'U' or 'L'. The default value is 'U'.

jobz

Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

```
call sspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call dspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call spgvx(ap, bp, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

*A*x* = λ*B*x*, *A*B*x* = λ*x*, or *B*A*x* = λ*x*.

Here *A* and *B* are assumed to be symmetric, stored in packed format, and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is *A*x = lambda*B*x*;

if *itype* = 2, the problem type is *A*B*x = lambda*x*;

if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz — CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *lambda(i)* in the half-open interval:

vl<*lambda*(*i*)≤*vu*.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo — CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

ap, *bp*, *work* — REAL for sspgvx

DOUBLE PRECISION for dspgvx.

Arrays:

ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.

The size of *ap* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*.

The size of *bp* must be at least max(1, *n**(*n*+1)/2).

work(*) is a workspace array, size at least max(1, 8*n*).

vl, *vu* — REAL for sspgvx

DOUBLE PRECISION for dspgvx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl*< *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* — INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0

if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for sspgvx

DOUBLE PRECISION for dspgvx.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array *z*. Constraints:

ldz≥ 1; if *jobz* = 'V', *ldz*≥ max(1, *n*) .

iwork

INTEGER.

Workspace array, size at least max(1, 5*n*).

Output Parameters

ap

On exit, the contents of *ap* are overwritten.

bp

On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$, in the same storage format as *B*.

m

INTEGER. The total number of eigenvalues found,

$0 \le m \le n$. If *range* = 'A', *m* = *n*, and if *range* = 'I',

m = *iu*-*il*+1.

w, *z*

REAL for sspgvx

DOUBLE PRECISION for dspgvx.

Arrays:

w(*), size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*) .

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows:

if *itype* = 1 or 2, Z^T*B*Z = I;

if *itype* = 3, Z^T*inv(*B*)*Z = I;

If *jobz* = 'N', then *z* is not referenced.

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail

INTEGER.

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, spptrf/dpptrf and sspevx/dspevx returned an error code:

If *info* = *i*≤*n*, sspevx/dspevx failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine spgvx interface are the following:

ap Holds the array *A* of size (*n**(*n*+1)/2).

bp Holds the array *B* of size (*n**(*n*+1)/2).

w Holds the vector with the number of elements *n*.

z Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.

ifail Holds the vector with the number of elements *n*.

itype Must be 1, 2, or 3. The default value is 1.

uplo Must be 'U' or 'L'. The default value is 'U'.

vl Default value for this element is *vl* = -HUGE(*vl*).

vu Default value for this element is *vu* = HUGE(*vl*).

il Default value for this argument is *il* = 1.

iu Default value for this argument is *iu* = *n*.

abstol Default value for this element is *abstol* = 0.0_WP.

jobz Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

Note that there will be an error condition if *ifail* is present and *z* is omitted.

range Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:

range = 'V', if one of or both *vl* and *vu* are present,

range = 'I', if one of or both *il* and *iu* are present,

range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ is used instead, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, set *abstol* to 2*?lamch('S').

?hpgvx

Computes selected eigenvalues and, optionally, eigenvectors of a generalized Hermitian positive-definite eigenproblem with matrices in packed storage.

Syntax

```
call chpgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call zhpgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call hpgvx(ap, bp, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

*A*x* = λ*B*x*, *A*B*x* = λ*x*, or *B*A*x* = λ*x*.

Here *A* and *B* are assumed to be Hermitian, stored in packed format, and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is *A*x = lambda*B*x*;

if *itype* = 2, the problem type is *A*B*x = lambda*x*;

if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *lambda(i)* in the half-open interval:

$vl < lambda(i) \leq vu$.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, *bp*, *work* COMPLEX for chpgvx

DOUBLE COMPLEX for zhpgvx.

Arrays:

ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.

The dimension of *ap* must be at least max(1, $n*(n+1)/2$).

bp(*) contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*.

The dimension of *bp* must be at least max(1, $n*(n+1)/2$).

work(*) is a workspace array, size at least max(1, 2*n*).

vl, *vu* REAL for chpgvx

DOUBLE PRECISION for zhpgvx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$

if $n = 0$.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol REAL for chpgvx

DOUBLE PRECISION for zhpgvx.

The absolute error tolerance for the eigenvalues.

See *Application Notes* for more information.

ldz INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1. If *jobz* = 'V', *ldz*≥ max(1, *n*).

rwork REAL for chpgvx

DOUBLE PRECISION for zhpgvx.

Workspace array, size at least max(1, 7*n*).

iwork INTEGER.

Workspace array, size at least max(1, 5*n*).

Output Parameters

ap On exit, the contents of *ap* are overwritten.

bp On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$, in the same storage format as *B*.

m INTEGER. The total number of eigenvalues found,

$0 \le m \le n$. If *range* = 'A', *m* = *n*, and if *range* = 'I',

m = *iu*-*il*+1.

w REAL for chpgvx

DOUBLE PRECISION for zhpgvx.

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z COMPLEX for chpgvx

DOUBLE COMPLEX for zhpgvx.

Array z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows:

if *itype* = 1 or 2, Z^H*B*Z = I;

if *itype* = 3, Z^H*inv(*B*)*Z = I;

If *jobz* = 'N', then *z* is not referenced.

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail INTEGER.

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, cpptrf/zpptrf and chpevx/zhpevx returned an error code:

If *info* = *i*≤*n*, chpevx/zhpevx failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If *info* = *n* + *i*, for 1 ≤*i*≤*n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hpgvx interface are the following:

ap Holds the array *A* of size (*n**(*n*+1)/2).

bp Holds the array *B* of size (*n**(*n*+1)/2).

w Holds the vector with the number of elements *n*.

z Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.

ifail Holds the vector with the number of elements *n*.

itype Must be 1, 2, or 3. The default value is 1.

uplo Must be 'U' or 'L'. The default value is 'U'.

vl Default value for this element is *vl* = -HUGE(*vl*).

vu Default value for this element is *vu* = HUGE(*vl*).

il Default value for this argument is *il* = 1.

iu Default value for this argument is *iu* = *n*.

abstol Default value for this element is *abstol* = 0.0_WP.

jobz Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

Note that there will be an error condition if *ifail* is present and *z* is omitted.

range Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:

range = 'V', if one of or both *vl* and *vu* are present,

range = 'I', if one of or both *il* and *iu* are present,

range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?sbgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

```
call ssbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
call dsbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
call sbgv(ab, bb, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form *A*x* = λ*B*x*. Here *A* and *B* are assumed to be symmetric and banded, and *B* is also positive definite.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.

n INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

ka INTEGER. The number of super- or sub-diagonals in *A* (*ka*≥ 0).

kb INTEGER. The number of super- or sub-diagonals in *B* (*kb*≥ 0).

ab, *bb*, *work* REAL for ssbgv

DOUBLE PRECISION for dsbgv

Arrays:

ab(*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of the array *ab* must be at least max(1, *n*).

bb(*ldbb*,*) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least max(1, *n*).

work(*) is a workspace array, dimension at least max(1, 3*n*)

ldab INTEGER. The leading dimension of the array *ab*; must be at least *ka*+1 .

ldbb INTEGER. The leading dimension of the array *bb*; must be at least *kb*+1.

ldz INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1. If *jobz* = 'V', *ldz*≥ max(1, *n*).

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *S* from the split Cholesky factorization $B = S^T*S$, as returned by pbstf/pbstf.

w, *z* REAL for ssbgv

DOUBLE PRECISION for dsbgv

Arrays:

w(*), size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*) .

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T*B*Z = I$.

If *jobz* = 'N', then *z* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, and

if *i*≤*n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for 1 ≤*i*≤*n*, then pbstf/pbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sbgv interface are the following:

ab Holds the array *A* of size (*ka*+1,*n*).

bb Holds the array *B* of size (*kb*+1,*n*).

w Holds the vector with the number of elements *n*.

z Holds the matrix *Z* of size (*n*, *n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

jobz Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

?hbgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

Syntax

```
call chbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
call zhbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
call hbgv(ab, bb, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form *A***x* = λ**B***x*. Here *A* and *B* are Hermitian and banded matrices, and matrix *B* is also positive definite.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo — CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

ka — INTEGER. The number of super- or sub-diagonals in *A*

(*ka*≥ 0).

kb — INTEGER. The number of super- or sub-diagonals in *B* (*kb*≥ 0).

ab, *bb*, *work* — COMPLEX for chbgv

DOUBLE COMPLEX for zhbgv

Arrays:

ab(*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of the array *ab* must be at least max(1, *n*).

bb(*ldbb*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least max(1, *n*).

work(*) is a workspace array, dimension at least max(1, *n*).

ldab — INTEGER. The leading dimension of the array *ab*; must be at least *ka*+1.

ldbb — INTEGER. The leading dimension of the array *bb*; must be at least *kb*+1.

ldz — INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1. If *jobz* = 'V', *ldz*≥ max(1, *n*).

rwork — REAL for chbgv

DOUBLE PRECISION for zhbgv.

Workspace array, size at least max(1, 3*n*).

Output Parameters

ab — On exit, the contents of *ab* are overwritten.

bb — On exit, contains the factor *S* from the split Cholesky factorization $B = S^H*S$, as returned by pbstf/pbstf.

w — REAL for chbgv

DOUBLE PRECISION for zhbgv.

Array, size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z — COMPLEX for chbgv

DOUBLE COMPLEX for zhbgv

Array z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H*B*Z = I$.

If *jobz* = 'N', then *z* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, and

if $i \le n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for $1 \le i \le n$, then pbstf/pbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hbgv interface are the following:

ab Holds the array *A* of size (*ka*+1,*n*).

bb Holds the array *B* of size (*kb*+1,*n*).

w Holds the vector with the number of elements *n*.

z Holds the matrix *Z* of size (*n*, *n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

jobz Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

?sbgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
call ssbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, iwork, liwork, info)
```

```
call dsbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, iwork, liwork, info)
```

```
call sbgvd(ab, bb, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form *A*x = λ*B*x*. Here *A* and *B* are assumed to be symmetric and banded, and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* (*n*≥ 0).

ka

INTEGER. The number of super- or sub-diagonals in *A*

(*ka*≥ 0).

kb

INTEGER. The number of super- or sub-diagonals in *B* (*kb*≥ 0).

ab, *bb*, *work*

REAL for ssbgvd

DOUBLE PRECISION for dsbgvd

Arrays:

ab(*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of the array *ab* must be at least max(1, *n*).

bb(*ldbb*,*) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

ldab

INTEGER. The leading dimension of the array *ab*; must be at least *ka*+1.

ldbb

INTEGER. The leading dimension of the array *bb*; must be at least *kb*+1.

ldz

INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1. If *jobz* = 'V', *ldz*≥ max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

If *n*≤ 1, *lwork*≥ 1;

If *jobz* = 'N' and *n*>1, *lwork*≥ 3*n*;

If *jobz* = 'V' and *n*>1, *lwork*≥ $2n^2+5n+1$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork

INTEGER.

Workspace array, its dimension max(1, *liwork*).

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

If *n*≤ 1, *liwork*≥ 1;

If *jobz* = 'N' and *n*>1, *liwork*≥ 1;

If *jobz* = 'V' and *n*>1, *liwork*≥ 5*n*+3.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

ab

On exit, the contents of *ab* are overwritten.

bb

On exit, contains the factor *S* from the split Cholesky factorization $B = S^T*S$, as returned by pbstf/pbstf.

w, *z*

REAL for ssbgvd

DOUBLE PRECISION for dsbgvd

Arrays:

w(*), size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T*B*Z = I$.

If *jobz* = 'N', then *z* is not referenced.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

iwork(1)	On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th argument had an illegal value. If *info* > 0, and if *i*≤*n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; if *info* = *n* + *i*, for 1 ≤*i*≤*n*, then pbstf/pbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sbgvd interface are the following:

ab	Holds the array *A* of size (*ka+1,n*).
bb	Holds the array *B* of size (*kb+1,n*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *work* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hbgvd
Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

`call chbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)`

`call zhbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)`

`call hbgvd(ab, bb, w [,uplo] [,z] [,info])`

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A*x = \lambda*B*x$. Here *A* and *B* are assumed to be Hermitian and banded, and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

jobz
`CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

uplo
`CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *ab* and *bb* store the upper triangles of *A* and *B*;
If *uplo* = `'L'`, arrays *ab* and *bb* store the lower triangles of *A* and *B*.

n
`INTEGER`. The order of the matrices *A* and *B* ($n \geq 0$).

ka
`INTEGER`. The number of super- or sub-diagonals in *A* ($ka \geq 0$).

kb
`INTEGER`. The number of super- or sub-diagonals in *B* ($kb \geq 0$).

ab, *bb*, *work*
`COMPLEX` for `chbgvd`
`DOUBLE COMPLEX` for `zhbgvd`
Arrays:
ab(*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.
The second dimension of the array *ab* must be at least max(1, *n*).
bb(*ldbb*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least max(1, *n*).

work is a workspace array, its dimension `max(1, lwork)`.

ldab — `INTEGER`. The leading dimension of the array *ab*; must be at least *ka*+1.

ldbb — `INTEGER`. The leading dimension of the array *bb*; must be at least *kb*+1.

ldz — `INTEGER`. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

lwork — `INTEGER`.

The dimension of the array *work*.

Constraints:

If $n \leq 1$, $lwork \geq 1$;

If *jobz* = 'N' and $n > 1$, $lwork \geq n$;

If *jobz* = 'V' and $n > 1$, $lwork \geq 2n^2$.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

rwork — `REAL` for `chbgvd`

`DOUBLE PRECISION` for `zhbgvd`.

Workspace array, size `max(1, lrwork)`.

lrwork — `INTEGER`.

The dimension of the array *rwork*.

Constraints:

If $n \leq 1$, $lrwork \geq 1$;

If *jobz* = 'N' and $n > 1$, $lrwork \geq n$;

If *jobz* = 'V' and $n > 1$, $lrwork \geq 2n^2+5n+1$.

If *lrwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

iwork — `INTEGER`.

Workspace array, size `max(1, liwork)`.

liwork — `INTEGER`.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $lwork \geq 1$;

If *jobz* = 'N' and $n > 1$, $liwork \geq 1$;

If *jobz* = 'V' and *n*>1, *liwork*≥ 5*n*+3.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See *Application Notes* for details.

Output Parameters

ab

On exit, the contents of *ab* are overwritten.

bb

On exit, contains the factor *S* from the split Cholesky factorization $B = S^H*S$, as returned by pbstf/pbstf.

w

REAL for chbgvd

DOUBLE PRECISION for zhbgvd.

Array, size at least max(1, *n*) .

If *info* = 0, contains the eigenvalues in ascending order.

z

COMPLEX for chbgvd

DOUBLE COMPLEX for zhbgvd

Array z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H*B*Z = I$.

If *jobz* = 'N', then *z* is not referenced.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork(1)

On exit, if *info* = 0, then *rwork*(1) returns the required minimal size of *lrwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, and

if *i*≤*n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for 1 ≤*i*≤*n*, then pbstf/pbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `hbgvd` interface are the following:

ab	Holds the array *A* of size (*ka*+*1*,*n*).
bb	Holds the array *B* of size (*kb*+*1*,*n*).
w	Holds the vector with the number of elements *n*.
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sbgvx
Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

```
call ssbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call dsbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call sbgvx(ab, bb, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A*x = \lambda*B*x$. Here *A* and *B* are assumed to be symmetric and banded, and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range

CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval:

$vl < lambda(i) \le vu$.

If *range* = 'I', the routine computes eigenvalues in range *il* to *iu*.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* ($n \ge 0$).

ka

INTEGER. The number of super- or sub-diagonals in *A*

($ka \ge 0$).

kb

INTEGER. The number of super- or sub-diagonals in *B* ($kb \ge 0$).

ab, *bb*, *work*

REAL for ssbgvx

DOUBLE PRECISION for dsbgvx

Arrays:

ab(*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of the array *ab* must be at least max(1, *n*).

bb(*ldbb*,*) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least max(1, *n*).

work(*) is a workspace array, size (7**n*).

ldab

INTEGER. The leading dimension of the array *ab*; must be at least *ka*+1.

ldbb

INTEGER. The leading dimension of the array *bb*; must be at least *kb*+1.

vl, *vu*

REAL for ssbgvx

DOUBLE PRECISION for dsbgvx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl*< *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0

if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for ssbgvx

DOUBLE PRECISION for dsbgvx.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1. If *jobz* = 'V', *ldz*≥ max(1, *n*).

ldq

INTEGER. The leading dimension of the output array *q*; *ldq* < 1.

If *jobz* = 'V', *ldq* < max(1, *n*).

iwork

INTEGER.

Workspace array, size (5**n*).

Output Parameters

ab

On exit, the contents of *ab* are overwritten.

bb

On exit, contains the factor *S* from the split Cholesky factorization $B = S^T*S$, as returned by pbstf/pbstf.

m

INTEGER. The total number of eigenvalues found,

0 ≤*m*≤*n*. If *range* = 'A', *m* = *n*, and if *range* = 'I',

m = *iu*-*il*+1.

w, *z*, *q*

REAL for ssbgvx

DOUBLE PRECISION for dsbgvx

Arrays:

w(*), size at least max(1, *n*) .

If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*) .

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T*B*Z = I$.

If *jobz* = 'N', then *z* is not referenced.

q(*ldq*,*) .

The second dimension of *q* must be at least max(1, *n*).

If *jobz* = 'V', then *q* contains the *n*-by-*n* matrix used in the reduction of *A*x = lambda*B*x* to standard form, that is, *C*x= lambda*x* and consequently *C* to tridiagonal form.

If *jobz* = 'N', then *q* is not referenced.

ifail INTEGER.

Array, size (*m*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, and

if $i \le n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for $1 \le i \le n$, then pbstf/pbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine sbgvx interface are the following:

ab Holds the array *A* of size (*ka*+1,*n*).

bb Holds the array *B* of size (*kb*+1,*n*).

w Holds the vector with the number of elements *n*.

z Holds the matrix *Z* of size (*n*, *n*).

ifail Holds the vector with the number of elements *n*.

q Holds the matrix *Q* of size (*n*, *n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

vl Default value for this element is *vl* = -HUGE(*vl*).

vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted. Note that there will be an error condition if *ifail* or *q* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hbgvx
Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

Syntax

call chbgvx(*jobz*, *range*, *uplo*, *n*, *ka*, *kb*, *ab*, *ldab*, *bb*, *ldbb*, *q*, *ldq*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *rwork*, *iwork*, *ifail*, *info*)

call zhbgvx(*jobz*, *range*, *uplo*, *n*, *ka*, *kb*, *ab*, *ldab*, *bb*, *ldbb*, *q*, *ldq*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *rwork*, *iwork*, *ifail*, *info*)

call hbgvx(*ab*, *bb*, *w* [,*uplo*] [,*z*] [,*vl*] [,*vu*] [,*il*] [,*iu*] [,*m*] [,*ifail*] [,*q*] [,*abstol*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A*x = \lambda*B*x$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz

CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range

CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval:

$vl < lambda(i) \le vu$.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* ($n \ge 0$).

ka

INTEGER. The number of super- or sub-diagonals in *A*

($ka \ge 0$).

kb

INTEGER. The number of super- or sub-diagonals in *B* ($kb \ge 0$).

ab, *bb*, *work*

COMPLEX for chbgvx

DOUBLE COMPLEX for zhbgvx

Arrays:

ab(*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of the array *ab* must be at least max(1, *n*).

bb(*ldbb*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least max(1, *n*).

work(*) is a workspace array, size at least max(1, *n*).

ldab

INTEGER. The leading dimension of the array *ab*; must be at least *ka*+1.

ldbb

INTEGER. The leading dimension of the array *bb*; must be at least *kb*+1.

vl, *vu*

REAL for chbgvx

DOUBLE PRECISION for zhbgvx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl*< *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤*il*≤*iu*≤*n*, if *n* > 0; *il*=1 and *iu*=0

if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for chbgvx

DOUBLE PRECISION for zhbgvx.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array *z*; *ldz*≥ 1. If *jobz* = 'V', *ldz*≥ max(1, *n*).

ldq

INTEGER. The leading dimension of the output array *q*; *ldq*≥ 1. If *jobz* = 'V', *ldq*≥ max(1, *n*).

rwork

REAL for chbgvx

DOUBLE PRECISION for zhbgvx.

Workspace array, size at least max(1, 7*n*).

iwork

INTEGER.

Workspace array, size at least max(1, 5*n*).

Output Parameters

ab

On exit, the contents of *ab* are overwritten.

bb

On exit, contains the factor *S* from the split Cholesky factorization $B = S^H*S$, as returned by pbstf/pbstf.

m

INTEGER. The total number of eigenvalues found,

0 ≤*m*≤*n*. If *range* = 'A', *m* = *n*, and if *range* = 'I',

m = *iu*-*il*+1.

w

REAL for chbgvx

DOUBLE PRECISION for zhbgvx.

Array *w*(*), size at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z, *q*

COMPLEX for chbgvx

DOUBLE COMPLEX for zhbgvx

Arrays:

z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H*B*Z = I$.

If *jobz* = 'N', then *z* is not referenced.

q(*ldq*,*).

The second dimension of *q* must be at least max(1, *n*).

If *jobz* = 'V', then *q* contains the *n*-by-*n* matrix used in the reduction of $Ax = \lambda Bx$ to standard form, that is, $Cx = \lambda x$ and consequently *C* to tridiagonal form.

If *jobz* = 'N', then *q* is not referenced.

ifail INTEGER.

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, and

if $i \le n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for $1 \le i \le n$, then pbstf/pbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine hbgvx interface are the following:

ab Holds the array *A* of size (*ka*+1,*n*).

bb Holds the array *B* of size (*kb*+1,*n*).

w Holds the vector with the number of elements *n*.

z Holds the matrix *Z* of size (*n*, *n*).

ifail Holds the vector with the number of elements *n*.

q Holds the matrix *Q* of size (*n*, *n*).

uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted. Note that there will be an error condition if *ifail* or *q* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol*+ε*max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

Generalized Nonsymmetric Eigenvalue Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving generalized nonsymmetric eigenproblems. See also computational routines that can be called to solve these problems. Table "Driver Routines for Solving Generalized Nonsymmetric Eigenproblems" lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Generalized Nonsymmetric Eigenproblems

Routine Name	Operation performed
gges	Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.
ggesx	Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.
gges3	Computes generalized Schur factorization for a pair of matrices.
ggev	Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.

Routine Name	Operation performed
ggevx	Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.
ggev3	Computes generalized Schur factorization for a pair of matrices.

?gges
Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.

Syntax

```
call sgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)
call dgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)
call cgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)
call zgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)
call gges(a, b, alphar, alphai, beta [,vsl] [,vsr] [,select] [,sdim] [,info])
call gges(a, b, alpha, beta [, vsl] [,vsr] [,select] [,sdim] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

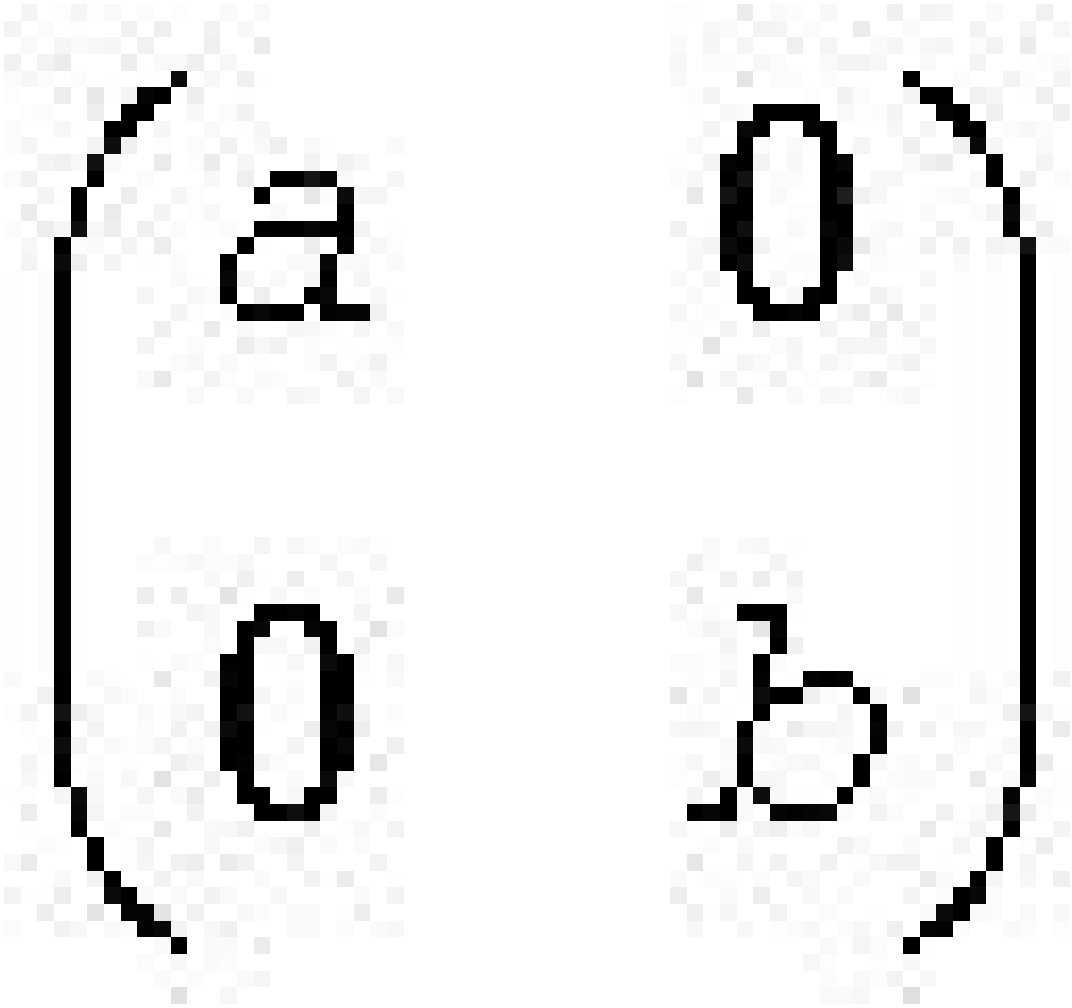
The ?gges routine computes the generalized eigenvalues, the generalized real/complex Schur form (*S*,*T*), optionally, the left and/or right matrices of Schur vectors (*vsl* and *vsr*) for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A*,*B*). This gives the generalized Schur factorization

$(A,B) = (vsl*S*vsr^H, vsl*T*vsr^H)$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix *S* and the upper triangular matrix *T*. The leading columns of *vsl* and *vsr* then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

If only the generalized eigenvalues are needed, use the driver ggev instead, which is faster.

A generalized eigenvalue for a pair of matrices (*A*,*B*) is a scalar *w* or a ratio *alpha* / *beta* = *w*, such that *A* - *w*B* is singular. It is usually represented as the pair (*alpha*, *beta*), as there is a reasonable interpretation for *beta*=0 or for both being zero. A pair of matrices (*S*,*T*) is in the generalized real Schur form if *T* is upper triangular with non-negative diagonal and *S* is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of *S* are "standardized" by making the corresponding elements of *T* have the form:



and the pair of corresponding 2-by-2 blocks in *S* and *T* will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (*S*,*T*) is in generalized complex Schur form if *S* and *T* are upper triangular and, in addition, the diagonal of *T* are non-negative real numbers.

The `?gges` routine replaces the deprecated `?gegs` routine.

Input Parameters

jobvsl

`CHARACTER*1`. Must be `'N'` or `'V'`.

If *jobvsl* = `'N'`, then the left Schur vectors are not computed.

If *jobvsl* = `'V'`, then the left Schur vectors are computed.

jobvsr

`CHARACTER*1`. Must be `'N'` or `'V'`.

If *jobvsr* = `'N'`, then the right Schur vectors are not computed.

If *jobvsr* = `'V'`, then the right Schur vectors are computed.

sort

`CHARACTER*1`. Must be `'N'` or `'S'`. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.

If *sort* = `'N'`, then eigenvalues are not ordered.

If *sort* = `'S'`, eigenvalues are ordered (see *selctg*).

selctg

LOGICAL FUNCTION of three REAL arguments for real flavors.

LOGICAL FUNCTION of two COMPLEX arguments for complex flavors.

selctg must be declared EXTERNAL in the calling subroutine.

If *sort* = 'S', *selctg* is used to select eigenvalues to sort to the top left of the Schur form.

If *sort* = 'N', *selctg* is not referenced.

For real flavors:

An eigenvalue (*alphar*(j) + *alphai*(j))/*beta*(j) is selected if *selctg*(*alphar*(j), *alphai*(j), *beta*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy *selctg*(*alphar*(j), *alphai*(j), *beta*(j)) = .TRUE. after ordering. In this case *info* is set to *n*+2 .

For complex flavors:

An eigenvalue *alpha*(j) / *beta*(j) is selected if *selctg*(*alpha*(j), *beta*(j)) is true.

Note that a selected complex eigenvalue may no longer satisfy *selctg*(*alpha*(j), *beta*(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* is set to *n*+2 (see *info* below).

n

INTEGER. The order of the matrices *A*, *B*, *vsl*, and *vsr* (*n*≥ 0).

a, *b*, *work*

REAL for sgges

DOUBLE PRECISION for dgges

COMPLEX for cgges

DOUBLE COMPLEX for zgges.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).

ldb

INTEGER. The leading dimension of the array *b*. Must be at least max(1, *n*).

ldvsl, *ldvsr*

INTEGER. The leading dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints:

ldvsl≥ 1. If *jobvsl* = 'V', *ldvsl*≥ max(1, *n*).

ldvsr≥ 1. If *jobvsr* = 'V', *ldvsr*≥ max(1, *n*).

lwork INTEGER.

The dimension of the array *work*.

lwork≥ max(1, 8*n*+16) for real flavors;

lwork≥ max(1, 2*n*) for complex flavors.

For good performance, *lwork* must generally be larger.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork REAL for cgges

DOUBLE PRECISION for zgges

Workspace array, size at least max(1, 8*n*).

This array is used in complex flavors only.

bwork LOGICAL.

Workspace array, size at least max(1, *n*).

Not referenced if *sort* = 'N'.

Output Parameters

a On exit, this array has been overwritten by its generalized Schur form *S*.

b On exit, this array has been overwritten by its generalized Schur form *T*.

sdim INTEGER.

If *sort* = 'N', *sdim*= 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *selctg* is true.

Note that for real flavors complex conjugate pairs for which *selctg* is true for either eigenvalue count as 2.

alphar, *alphai* REAL for sgges;

DOUBLE PRECISION for dgges.

Arrays, size at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha COMPLEX for cgges;

DOUBLE COMPLEX for zgges.

Array, size at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta REAL for sgges

DOUBLE PRECISION for dgges

COMPLEX for cgges

`DOUBLE COMPLEX` for `zgges`.

Array, size at least max(1, *n*).

For real flavors:

On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,..., *n*, will be the generalized eigenvalues.

alphar(j) + *alphai*(j)*i and *beta*(j), j=1,..., *n* are the diagonals of the complex Schur form (*S*,*T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A*,*B*) were further reduced to triangular form using complex unitary transformations. If *alphai*(j) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.

For complex flavors:

On exit, *alpha*(j)/*beta*(j), j=1,..., *n*, will be the generalized eigenvalues. *alphaalpha*(j) and *beta*(j), j=1,..., *n* are the diagonals of the complex Schur form (*S*,*T*) output by `cgges`/`zgges`. The *beta*(j) will be non-negative real.

See also *Application Notes* below.

vsl, *vsr*

`REAL` for `sgges`

`DOUBLE PRECISION` for `dgges`

`COMPLEX` for `cgges`

`DOUBLE COMPLEX` for `zgges`.

Arrays:

vsl(*ldvsl*,*), the second dimension of *vsl* must be at least max(1, *n*).

If *jobvsl* = 'V', this array will contain the left Schur vectors.

If *jobvsl* = 'N', *vsl* is not referenced.

vsr(*ldvsr*,*), the second dimension of *vsr* must be at least max(1, *n*).

If *jobvsr* = 'V', this array will contain the right Schur vectors.

If *jobvsr* = 'N', *vsr* is not referenced.

work(1)

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info

`INTEGER`.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, and

i≤*n*:

the *QZ* iteration failed. (*A*, *B*) is not in Schur form, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info* +1,..., *n* should be correct.

i > *n*: errors that usually indicate LAPACK problems:

i = *n*+1: other than *QZ* iteration failed in hgeqz;

`i = n+2`: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy `selctg = .TRUE.`. This could also be caused due to scaling;

`i = n+3`: reordering failed in tgsen.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `gges` interface are the following:

`a`	Holds the matrix *A* of size (*n*, *n*).
`b`	Holds the matrix *B* of size (*n*, *n*).
`alphar`	Holds the vector of length *n*. Used in real flavors only.
`alphai`	Holds the vector of length *n*. Used in real flavors only.
`alpha`	Holds the vector of length *n*. Used in complex flavors only.
`beta`	Holds the vector of length *n*.
`vsl`	Holds the matrix *VSL* of size (*n*, *n*).
`vsr`	Holds the matrix *VSR* of size (*n*, *n*).
`jobvsl`	Restored based on the presence of the argument *vsl* as follows: `jobvsl = 'V'`, if *vsl* is present, `jobvsl = 'N'`, if *vsl* is omitted.
`jobvsr`	Restored based on the presence of the argument *vsr* as follows: `jobvsr = 'V'`, if *vsr* is present, `jobvsr = 'N'`, if *vsr* is omitted.
`sort`	Restored based on the presence of the argument *select* as follows: `sort = 'S'`, if *select* is present, `sort = 'N'`, if *select* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?ggesx

Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.

Syntax

```
call sggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, iwork, liwork, bwork, info)
```

```
call dggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, iwork, liwork, bwork, info)
```

```
call cggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork, iwork, liwork, bwork, info)
```

```
call zggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork, iwork, liwork, bwork, info)
```

```
call ggesx(a, b, alphar, alphai, beta [,vsl] [,vsr] [,select] [,sdim] [,rconde] [,rcondv] [,info])
```

```
call ggesx(a, b, alpha, beta [, vsl] [,vsr] [,select] [,sdim] [,rconde] [,rcondv] [,info])
```

Include Files

- mkl.fi, lapack.f90

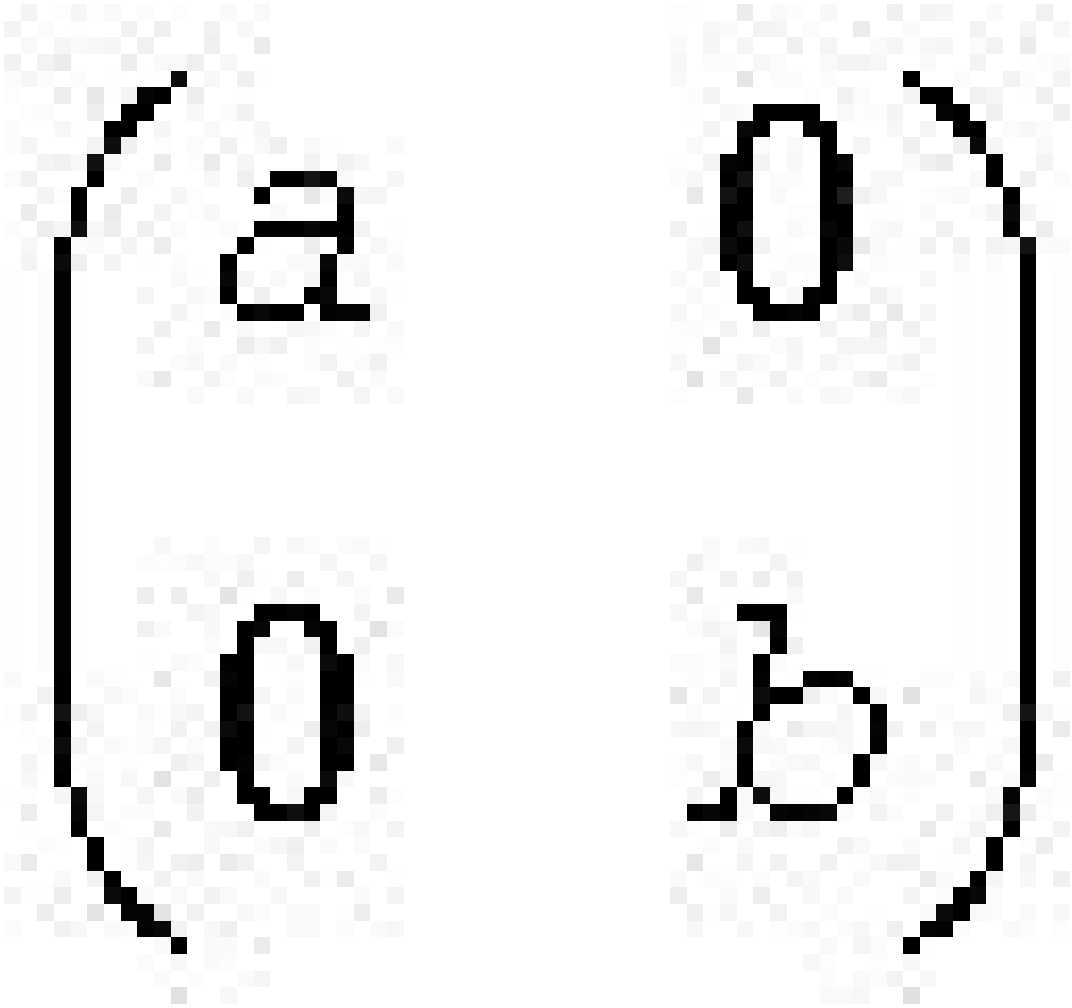
Description

The routine computes for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A*,*B*), the generalized eigenvalues, the generalized real/complex Schur form (*S*,*T*), optionally, the left and/or right matrices of Schur vectors (*vsl* and *vsr*). This gives the generalized Schur factorization

$(A,B) = (vsl*S*vsr^H, vsl*T*vsr^H)$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix *S* and the upper triangular matrix *T*; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues (*rcondv*). The leading columns of *vsl* and *vsr* then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (*A*,*B*) is a scalar *w* or a ratio $alpha / beta = w$, such that $A - w*B$ is singular. It is usually represented as the pair (*alpha*, *beta*), as there is a reasonable interpretation for $beta=0$ or for both being zero. A pair of matrices (*S*,*T*) is in generalized real Schur form if *T* is upper triangular with non-negative diagonal and *S* is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of *S* will be "standardized" by making the corresponding elements of *T* have the form:



and the pair of corresponding 2-by-2 blocks in *S* and *T* will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (*S*,*T*) is in generalized complex Schur form if *S* and *T* are upper triangular and, in addition, the diagonal of *T* are non-negative real numbers.

Input Parameters

jobvsl

`CHARACTER*1`. Must be `'N'` or `'V'`.

If *jobvsl* = `'N'`, then the left Schur vectors are not computed.

If *jobvsl* = `'V'`, then the left Schur vectors are computed.

jobvsr

`CHARACTER*1`. Must be `'N'` or `'V'`.

If *jobvsr* = `'N'`, then the right Schur vectors are not computed.

If *jobvsr* = `'V'`, then the right Schur vectors are computed.

sort

`CHARACTER*1`. Must be `'N'` or `'S'`. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.

If *sort* = `'N'`, then eigenvalues are not ordered.

If *sort* = `'S'`, eigenvalues are ordered (see *selctg*).

selctg

`LOGICAL FUNCTION` of three `REAL` arguments for real flavors.

LOGICAL FUNCTION of two COMPLEX arguments for complex flavors.

selctg must be declared EXTERNAL in the calling subroutine.

If *sort* = 'S', *selctg* is used to select eigenvalues to sort to the top left of the Schur form.

If *sort* = 'N', *selctg* is not referenced.

For real flavors:

An eigenvalue (*alphar*(j) + *alphai*(j))/*beta*(j) is selected if *selctg*(*alphar*(j), *alphai*(j), *beta*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy *selctg*(*alphar*(j), *alphai*(j), *beta*(j)) = .TRUE. after ordering. In this case *info* is set to *n*+2.

For complex flavors:

An eigenvalue *alpha*(j) / *beta*(j) is selected if *selctg*(*alpha*(j), *beta*(j)) is true.

Note that a selected complex eigenvalue may no longer satisfy *selctg*(*alpha*(j), *beta*(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* is set to *n*+2 (see *info* below).

sense — CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If *sense* = 'N', none are computed;

If *sense* = 'E', computed for average of selected eigenvalues only;

If *sense* = 'V', computed for selected deflating subspaces only;

If *sense* = 'B', computed for both.

If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.

n — INTEGER. The order of the matrices *A*, *B*, *vsl*, and *vsr* (*n*≥ 0).

a, *b*, *work* — REAL for sggesx

DOUBLE PRECISION for dggesx

COMPLEX for cggesx

DOUBLE COMPLEX for zggesx.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The leading dimension of the array *a*.

Must be at least max(1, *n*).

ldb — INTEGER. The leading dimension of the array *b*.

Must be at least max(1, *n*).

ldvsl, *ldvsr* — INTEGER. The leading dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints:

ldvsl≥ 1. If *jobvsl* = 'V', *ldvsl*≥ max(1, *n*).

ldvsr≥ 1. If *jobvsr* = 'V', *ldvsr*≥ max(1, *n*).

lwork — INTEGER.

The dimension of the array *work*.

For real flavors:

If *n*=0 then *lwork*≥1.

If *n*>0 and *sense* = 'N', then *lwork*≥ max(8*n, 6*n+16).

If *n*>0 and *sense* = 'E', 'V', or 'B', then *lwork*≥ max(8*n, 6*n+16, 2**sdim**(n-sdim));

For complex flavors:

If *n*=0 then *lwork*≥1.

If *n*>0 and *sense* = 'N', then *lwork*≥ max(1, 2*n);

If *n*>0 and *sense* = 'E', 'V', or 'B', then *lwork*≥ max(1, 2*n, 2**sdim**(n-sdim)).

Note that 2**sdim**(n-sdim) ≤ n*n/2.

An error is only returned if *lwork*< max(8*n, 6*n+16) for real flavors, and *lwork*< max(1, 2*n) for complex flavors, but if *sense* = 'E', 'V', or 'B', this may not be large enough.

If *lwork*=-1, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla.

rwork — REAL for cggesx

DOUBLE PRECISION for zggesx

Workspace array, size at least max(1, 8*n*).

This array is used in complex flavors only.

iwork — INTEGER.

Workspace array, size max(1, *liwork*).

liwork — INTEGER.

The dimension of the array *iwork*.

If *sense* = 'N', or *n*=0, then *liwork*≥1,

otherwise *liwork*≥ (*n*+6) for real flavors, and *liwork*≥ (*n*+2) for complex flavors.

If *liwork*=-1, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla.

bwork

LOGICAL.

Workspace array, size at least max(1, *n*).

Not referenced if *sort* = 'N'.

Output Parameters

a

On exit, this array has been overwritten by its generalized Schur form *S*.

b

On exit, this array has been overwritten by its generalized Schur form *T*.

sdim

INTEGER.

If *sort* = 'N', *sdim*= 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *selctg* is true.

Note that for real flavors complex conjugate pairs for which *selctg* is true for either eigenvalue count as 2.

alphar, *alphai*

REAL for sggesx;

DOUBLE PRECISION for dggesx.

Arrays, size at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha

COMPLEX for cggesx;

DOUBLE COMPLEX for zggesx.

Array, size at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

REAL for sggesx

DOUBLE PRECISION for dggesx

COMPLEX for cggesx

DOUBLE COMPLEX for zggesx.

Array, size at least max(1, *n*).

For real flavors:

On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,..., *n* will be the generalized eigenvalues.

alphar(j) + *alphai*(j)*i and *beta*(j), j=1,..., *n* are the diagonals of the complex Schur form (*S*,*T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A*,*B*) were further reduced to triangular form using complex unitary transformations. If *alphai*(j) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.

For complex flavors:

On exit, *alpha*(j)/*beta*(j), j=1,..., *n* will be the generalized eigenvalues. *alpha*(j) and *beta*(j), j=1,..., *n* are the diagonals of the complex Schur form (*S*,*T*) output by `cggesx`/`zggesx`. The *beta*(j) will be non-negative real.

See also *Application Notes* below.

vsl, *vsr*

`REAL` for `sggesx`

`DOUBLE PRECISION` for `dggesx`

`COMPLEX` for `cggesx`

`DOUBLE COMPLEX` for `zggesx`.

Arrays:

vsl(*ldvsl*,*), the second dimension of *vsl* must be at least max(1, *n*).

If *jobvsl* = `'V'`, this array will contain the left Schur vectors.

If *jobvsl* = `'N'`, *vsl* is not referenced.

vsr(*ldvsr*,*), the second dimension of *vsr* must be at least max(1, *n*).

If *jobvsr* = `'V'`, this array will contain the right Schur vectors.

If *jobvsr* = `'N'`, *vsr* is not referenced.

rconde, *rcondv*

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Arrays, size (2) each

If *sense* = `'E'` or `'B'`, *rconde*[0] and *rconde*[1] contain the reciprocal condition numbers for the average of the selected eigenvalues.

Not referenced if *sense* = `'N'` or `'V'`.

If *sense* = `'V'` or `'B'`, *rcondv*(1) and *rcondv*(2) contain the reciprocal condition numbers for the selected deflating subspaces.

Not referenced if *sense* = `'N'` or `'E'`.

work(1)

On exit, if *info* = `0`, then *work(1)* returns the required minimal size of *lwork*.

iwork(1)

On exit, if *info* = `0`, then *iwork(1)* returns the required minimal size of *liwork*.

info

`INTEGER`.

If *info* = `0`, the execution is successful.

If *info* = `-i`, the *i*th parameter had an illegal value.

If *info* = *i*, and

i≤*n*:

the *QZ* iteration failed. (*A*, *B*) is not in Schur form, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info* +1,..., *n* should be correct.

i > *n*: errors that usually indicate LAPACK problems:

i = *n*+1: other than *QZ* iteration failed in ?hgeqz;

i = *n*+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy *selctg* = .TRUE.. This could also be caused due to scaling;

i = *n*+3: reordering failed in tgsen.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ggesx interface are the following:

a Holds the matrix *A* of size (*n*, *n*).

b Holds the matrix *B* of size (*n*, *n*).

alphar Holds the vector of length *n*. Used in real flavors only.

alphai Holds the vector of length *n*. Used in real flavors only.

alpha Holds the vector of length *n*. Used in complex flavors only.

beta Holds the vector of length *n*.

vsl Holds the matrix *VSL* of size (*n*, *n*).

vsr Holds the matrix *VSR* of size (*n*, *n*).

rconde Holds the vector of length (2).

rcondv Holds the vector of length (2).

jobvsl Restored based on the presence of the argument *vsl* as follows:

jobvsl = 'V', if *vsl* is present,

jobvsl = 'N', if *vsl* is omitted.

jobvsr Restored based on the presence of the argument *vsr* as follows:

jobvsr = 'V', if *vsr* is present,

jobvsr = 'N', if *vsr* is omitted.

sort Restored based on the presence of the argument *select* as follows:

sort = 'S', if *select* is present,

sort = 'N', if *select* is omitted.

sense Restored based on the presence of arguments *rconde* and rcondv as follows:

sense = 'B', if both *rconde* and *rcondv* are present,

sense = 'E', if *rconde* is present and *rcondv* omitted,

sense = 'V', if *rconde* is omitted and *rcondv* present,

sense = 'N', if both *rconde* and *rcondv* are omitted.

Note that there will be an error condition if *rconde* or *rcondv* are present and *select* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?gges3
Computes generalized Schur factorization for a pair of matrices.

Syntax

```
call sgges3 (jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphar, alphai,
beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info )
```

```
call dgges3 (jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphar, alphai,
beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info )
```

```
call cgges3 (jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta, vsl,
ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info )
```

```
call zgges3 (jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta, vsl,
ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info )
```

Include Files

- mkl.fi

Description

For a pair of *n*-by-*n* real or complex nonsymmetric matrices (*A*,*B*), ?gges3 computes the generalized eigenvalues, the generalized real or complex Schur form (*S*,*T*), and optionally the left or right matrices of Schur vectors (*VSL* and *VSR*). This gives the generalized Schur factorization

(*A*,*B*) = ((*VSL*)**S**(*VSR*)^T, (*VSL*)**T**(*VSR*)^T) for real (*A*,*B*)

or

(*A*,*B*) = ((*VSL*)**S**(*VSR*)^H, (*VSL*)**T**(*VSR*)^H) for complex (*A*,*B*)

where (*VSR*)^H is the conjugate-transpose of *VSR*.

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix *S* and the upper triangular matrix *T*. The leading columns of *VSL* and *VSR* then form an orthonormal basis for the corresponding left and right eigenspaces (deflating subspaces).

NOTE
If only the generalized eigenvalues are needed, use the driver `?ggev` instead, which is faster.

A generalized eigenvalue for a pair of matrices (*A*,*B*) is a scalar *w* or a ratio *alpha*/*beta* = *w*, such that A - *w**B is singular. It is usually represented as the pair (*alpha*,*beta*), as there is a reasonable interpretation for *beta*=0 or both being zero.

For real flavors:

A pair of matrices (*S*,*T*) is in generalized real Schur form if *T* is upper triangular with non-negative diagonal and *S* is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of *S* will be "standardized" by making the corresponding elements of *T* have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in *S* and *T* have a complex conjugate pair of generalized eigenvalues.

For complex flavors:

A pair of matrices (*S*,*T*) is in generalized complex Schur form if *S* and *T* are upper triangular and, in addition, the diagonal elements of *T* are non-negative real numbers.

Input Parameters

jobvsl — CHARACTER*1. = 'N': do not compute the left Schur vectors;

jobvsr — CHARACTER*1. = 'N': do not compute the right Schur vectors;

= 'V': compute the right Schur vectors.

sort — CHARACTER*1. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.

= 'N': Eigenvalues are not ordered;

= 'S': Eigenvalues are ordered (see *selctg*).

selctg — LOGICAL. *selctg* is a function of three arguments for real flavors or two arguments for complex flavors. *selctg* must be declared EXTERNAL in the calling subroutine. If *sort* = 'N', *selctg* is not referenced. If *sort* = 'S', *selctg* is used to select eigenvalues to sort to the top left of the Schur form.

For real flavors:

An eigenvalue (*alphar(j)* + *alphai(j)*)/*beta(j)* is selected if *selctg*(*alphar(j)*,*alphai(j)*,*beta(j)*) is true. In other words, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy *selctg*(*alphar(j)*,*alphai(j)*, *beta(j)*) = .TRUE. after ordering. *info* is to be set to *n*+2 in this case.

For complex flavors:

An eigenvalue *alpha*(*j*)/*beta*(*j*) is selected if *selctg*(*alpha*(*j*),*beta*(*j*)) is true.

Note that a selected complex eigenvalue may no longer satisfy *selctg*(*alpha*(*j*),*beta*(*j*))= .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned), in this case *info* is set to *n* + 2 (See *info* below)..

n INTEGER. The order of the matrices *A*, *B*, *VSL*, and *VSR*. *n*≥ 0.

a REAL for sgges3

DOUBLE PRECISION for dgges3

COMPLEX for cgges3

DOUBLE COMPLEX for zgges3

Array, size (*lda*, *n*). On entry, the first of the pair of matrices.

lda INTEGER. The leading dimension of *a*. *lda*≥ max(1,*n*).

b REAL for sgges3

DOUBLE PRECISION for dgges3

COMPLEX for cgges3

DOUBLE COMPLEX for zgges3

Array, size (*ldb*, *n*). On entry, the second of the pair of matrices.

ldb INTEGER. The leading dimension of *b*. *ldb*≥ max(1,*n*).

ldvsl INTEGER. The leading dimension of the matrix *VSL*. *ldvsl*≥ 1, and if *jobvsl* = 'V', *ldvsl*≥ n.

ldvsr INTEGER. The leading dimension of the matrix *VSR*. *ldvsr*≥ 1, and if *jobvsr* = 'V', *ldvsr*≥ n.

lwork INTEGER. The size of the array *work*. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

work REAL for sgges3

DOUBLE PRECISION for dgges3

COMPLEX for cgges3

DOUBLE COMPLEX for zgges3

Array, size (MAX(1,*lwork*)).

On exit, if *info* = 0, *work*(1) returns the optimal lwork.

rwork REAL for cgges3

DOUBLE PRECISION for zgges3

Array, size (8**n*).

bwork LOGICAL. Array, size (*n*). Not referenced if *sort* = 'N'.

Output Parameters

a On exit, *a* is overwritten by its generalized Schur form S.

b On exit, *b* is overwritten by its generalized Schur form T.

sdim INTEGER. If *sort* = 'N', *sdim* = 0. If *sort* = 'S', *sdim* = number of eigenvalues (after sorting) for which *selctg* is true.

alpha `COMPLEX` for `cgges3`

`DOUBLE COMPLEX` for `zgges3`

Array, size (*n*).

alphar `REAL` for `sgges3`

`DOUBLE PRECISION` for `dgges3`

Array, size (*n*).

alphai `REAL` for `sgges3`

`DOUBLE PRECISION` for `dgges3`

Array, size (*n*).

beta `REAL` for `sgges3`

`DOUBLE PRECISION` for `dgges3`

`COMPLEX` for `cgges3`

`DOUBLE COMPLEX` for `zgges3`

Array, size (*n*).

For real flavors:

On exit, (*alphar*(*j*) + *alphai*(*j*)*i)/*beta*(*j*), j=1,...,*n*, are the generalized eigenvalues. *alphar*(*j*) + *alphai*(*j*)*i, and *beta*(*j*),j=1,...,*n* are the diagonals of the complex Schur form (*S*,*T*) that would result if the 2-by-2 diagonal blocks of the real Schur form of (*a*,*b*) were further reduced to triangular form using 2-by-2 complex unitary transformations. If *alphai*(*j*) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-st eigenvalues are a complex conjugate pair, with *alphai*(*j* + 1) negative.

Note: the quotients *alphar*(*j*)/*beta*(*j*) and *alphai*(*j*)/*beta*(*j*) can easily over- or underflow, and *beta*(*j*) might even be zero. Thus, you should avoid computing the ratio *alpha*/*beta* by simply dividing *alpha* by *beta*. However, *alphar* and *alphai* is always less than and usually comparable with norm(*a*) in magnitude, and *beta* is always less than and usually comparable with norm(*b*).

For complex flavors:

On exit, *alpha*(*j*)(*j*)/*beta*(*j*), j=1,...,*n*, are the generalized eigenvalues. *alpha*(*j*), j=1,...,*n* and *beta*(*j*), j=1,...,*n* are the diagonals of the complex Schur form (*a*,*b*) output by ?gges3. The *beta*(*j*) is non-negative real.

Note: the quotient *alpha*(*j*)/*beta*(*j*) can easily over- or underflow, and *beta*(*j*) might even be zero. Thus, you should avoid computing the ratio *alpha*/*beta* by simply dividing *alpha* by *beta*. However, *alpha* is always less than and usually comparable with norm(*a*) in magnitude, and *beta* is always less than and usually comparable with norm(*b*).

vsl

REAL for sgges3

DOUBLE PRECISION for dgges3

COMPLEX for cgges3

DOUBLE COMPLEX for zgges3

Array, size (*ldvsl*, *n*).

If *jobvsl* = 'V', *vsl* contains the left Schur vectors. Not referenced if *jobvsl* = 'N'.

vsr

REAL for sgges3

DOUBLE PRECISION for dgges3

COMPLEX for cgges3

DOUBLE COMPLEX for zgges3

Array, size (*ldvsr*, *n*).

If *jobvsr* = 'V', *vsr* contains the right Schur vectors. Not referenced if *jobvsr* = 'N'.

info

INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

=1,...,*n*:

- for real flavors:

  The QZ iteration failed. (*a*,*b*) are not in Schur form, but *alphar*(*j*), *alphai*(*j*) and *beta*(*j*) should be correct for *j*=*info* +1,...,*n*.
- for complex flavors:

  The QZ iteration failed. (*a*,*b*) are not in Schur form, but *alpha*(*j*) and *beta*(*j*) should be correct for *j*=*info*+1,...,*n*.

> *n*:

- =*n*+1: other than QZ iteration failed in ?hgeqz.
- =*n*+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Generalized Schur form no longer satisfy *selctg*= .TRUE. This could also be caused due to scaling.
- =*n*+3: reordering failed in ?tgsen.

?ggev

Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.

Syntax

```
call sggev(jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta, vl, ldvl, vr, ldvr, work, lwork, info)
```

```
call dggev(jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta, vl, ldvl, vr, ldvr, work, lwork, info)
```

```
call cggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
```

```
call zggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
```

```
call ggev(a, b, alphar, alphai, beta [,vl] [,vr] [,info])
```

```
call ggev(a, b, alpha, beta [, vl] [,vr] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The ?ggev routine computes the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A*,*B*).

A generalized eigenvalue for a pair of matrices (*A*,*B*) is a scalar λ or a ratio *alpha* / *beta* = λ, such that *A* – λ**B* is singular. It is usually represented as the pair (*alpha*, *beta*), as there is a reasonable interpretation for *beta* =0 and even for both being zero.

The right generalized eigenvector *v*(j) corresponding to the generalized eigenvalue λ(j) of (*A*,*B*) satisfies

*A***v*(j) = λ(j)**B***v*(j).

The left generalized eigenvector *u*(j) corresponding to the generalized eigenvalue λ(j) of (*A*,*B*) satisfies

$u(j)^H * A = \lambda(j) * u(j)^H * B$

where $u(j)^H$ denotes the conjugate transpose of *u*(j).

The ?ggev routine replaces the deprecated ?gegv routine.

Input Parameters

jobvl
: CHARACTER*1. Must be 'N' or 'V'.
If *jobvl* = 'N', the left generalized eigenvectors are not computed;
If *jobvl* = 'V', the left generalized eigenvectors are computed.

jobvr
: CHARACTER*1. Must be 'N' or 'V'.
If *jobvr* = 'N', the right generalized eigenvectors are not computed;
If *jobvr* = 'V', the right generalized eigenvectors are computed.

n
: INTEGER. The order of the matrices *A*, *B*, *vl*, and *vr* (*n*≥ 0).

a, *b*, *work*

REAL for sggev

DOUBLE PRECISION for dggev

COMPLEX for cggev

DOUBLE COMPLEX for zggev.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).

ldb

INTEGER. The leading dimension of the array *b*. Must be at least max(1, *n*).

ldvl, *ldvr*

INTEGER. The leading dimensions of the output matrices *vl* and *vr*, respectively.

Constraints:

ldvl ≥ 1. If *jobvl* = 'V', *ldvl* ≥ max(1, *n*).

ldvr ≥ 1. If *jobvr* = 'V', *ldvr* ≥ max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*.

lwork ≥ max(1, 8*n*+16) for real flavors;

lwork ≥ max(1, 2*n*) for complex flavors.

For good performance, *lwork* must generally be larger.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork

REAL for cggev

DOUBLE PRECISION for zggev

Workspace array, size at least max(1, 8*n*).

This array is used in complex flavors only.

Output Parameters

a, *b*

On exit, these arrays have been overwritten.

alphar, *alphai*

REAL for sggev;

DOUBLE PRECISION for dggev.

Arrays, size at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha

`COMPLEX` for `cggev`;

`DOUBLE COMPLEX` for `zggev`.

Array, size at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

`REAL` for `sggev`

`DOUBLE PRECISION` for `dggev`

`COMPLEX` for `cggev`

`DOUBLE COMPLEX` for `zggev`.

Array, size at least max(1, *n*).

For real flavors:

On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,..., *n*, are the generalized eigenvalues.

If *alphai*(j) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.

For complex flavors:

On exit, *alpha*(j)/*beta*(j), j=1,..., *n*, are the generalized eigenvalues.

See also *Application Notes* below.

vl, *vr*

`REAL` for `sggev`

`DOUBLE PRECISION` for `dggev`

`COMPLEX` for `cggev`

`DOUBLE COMPLEX` for `zggev`.

Arrays:

vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*). Contains the matrix of left generalized eigenvectors *VL*.

If $jobvl = 'V'$, the left generalized eigenvectors u_j are stored one after another in the columns of *VL*, in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has `abs(Re) + abs(Im) = 1`.

If $jobvl = 'N'$, *vl* is not referenced.

For real flavors:

If the j-th eigenvalue is real,then $u_j = VL_{*,j}$, the *j*-th column of *VL*.

If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for $i = \text{sqrt}(-1)$, $u_j = VL_{*,j} + i*VL_{*,j+1}$ and $u_{j+1} = VL_{*,j} - i*VL_{*,j+1}$.

For complex flavors:

$u_j = VL_{*,j}$, the *j*-th column of *vl*.

vr(*ldvr*,*); the second dimension of *vr* must be at least max(1, *n*). Contains the matrix of right generalized eigenvectors *VR*.

If *jobvr* = 'V', the right generalized eigenvectors v_j are stored one after another in the columns of *VR*, in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has abs(Re) + abs(Im) = 1.

If *jobvr* = 'N', *vr* is not referenced.

For real flavors:

If the *j*-th eigenvalue is real, then $v_j = VR_{*,j}$, the *j*-th column of *VR*.

If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then $v_j = VR_{*,j} + i*VR_{*,j+1}$ and $v_{j+1} = VR_{*,j} - i*VR_{*,j+1}$.

For complex flavors:

$v_j = VR_{*,j}$, the *j*-th column of *VR*.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and

i≤*n*: the *QZ* iteration failed. No eigenvectors have been calculated, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info*+1, ..., *n* should be correct.

i > *n*: errors that usually indicate LAPACK problems:

i = *n*+1: other than *QZ* iteration failed in hgeqz;

i = *n*+2: error return from tgevc.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ggev interface are the following:

a Holds the matrix *A* of size (*n*, *n*).

b Holds the matrix *B* of size (*n*, *n*).

alphar Holds the vector of length *n*. Used in real flavors only.

alphai Holds the vector of length *n*. Used in real flavors only.

alpha Holds the vector of length *n*. Used in complex flavors only.

beta Holds the vector of length *n*.

vl Holds the matrix *VL* of size (*n*, *n*).

vr Holds the matrix *VR* of size (*n*, *n*).

jobvl Restored based on the presence of the argument *vl* as follows:

jobvl = 'V', if *vl* is present,

jobvl = 'N', if *vl* is omitted.

jobvr Restored based on the presence of the argument *vr* as follows:

jobvr = 'V', if *vr* is present,

jobvr = 'N', if *vr* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?ggevx
Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

Syntax

call sggevx(*balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *b*, *ldb*, *alphar*, *alphai*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, *bbnrm*, *rconde*, *rcondv*, *work*, *lwork*, *iwork*, *bwork*, *info*)

call dggevx(*balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *b*, *ldb*, *alphar*, *alphai*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, *bbnrm*, *rconde*, *rcondv*, *work*, *lwork*, *iwork*, *bwork*, *info*)

call cggevx(*balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *b*, *ldb*, *alpha*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, *bbnrm*, *rconde*, *rcondv*, *work*, *lwork*, *rwork*, *iwork*, *bwork*, *info*)

call zggevx(*balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *b*, *ldb*, *alpha*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, *bbnrm*, *rconde*, *rcondv*, *work*, *lwork*, *rwork*, *iwork*, *bwork*, *info*)

call ggevx(*a*, *b*, *alphar*, *alphai*, *beta* [,*vl*] [,*vr*] [,*balanc*] [,*ilo*] [,*ihi*] [, *lscale*] [,*rscale*] [,*abnrm*] [,*bbnrm*] [,*rconde*] [,*rcondv*] [,*info*])

call ggevx(*a*, *b*, *alpha*, *beta* [, *vl*] [,*vr*] [,*balanc*] [,*ilo*] [,*ihi*] [,*lscale*] [, *rscale*] [,*abnrm*] [,*bbnrm*] [,*rconde*] [,*rcondv*] [,*info*])

Include Files

- mkl.fi, lapack.f90

Description

The routine computes for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A*,*B*), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, and *bbnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

A generalized eigenvalue for a pair of matrices (*A*,*B*) is a scalar λ or a ratio *alpha* / *beta* = λ, such that *A* – λ**B* is singular. It is usually represented as the pair (*alpha*, *beta*), as there is a reasonable interpretation for *beta*=0 and even for both being zero. The right generalized eigenvector *v*(j) corresponding to the generalized eigenvalue λ(j) of (*A*,*B*) satisfies

*A**v*(j) = λ(j)**B***v*(j).

The left generalized eigenvector *u*(j) corresponding to the generalized eigenvalue λ(j) of (*A*,*B*) satisfies

u(j)^H**A* = λ(j)**u*(j)^H**B*

where *u*(j)^H denotes the conjugate transpose of *u*(j).

Input Parameters

balanc

CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Specifies the balance option to be performed.

If *balanc* = 'N', do not diagonally scale or permute;

If *balanc* = 'P', permute only;

If *balanc* = 'S', scale only;

If *balanc* = 'B', both permute and scale.

Computed reciprocal condition numbers will be for the matrices after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

jobvl

CHARACTER*1. Must be 'N' or 'V'.

If *jobvl* = 'N', the left generalized eigenvectors are not computed;

If *jobvl* = 'V', the left generalized eigenvectors are computed.

jobvr

CHARACTER*1. Must be 'N' or 'V'.

If *jobvr* = 'N', the right generalized eigenvectors are not computed;

If *jobvr* = 'V', the right generalized eigenvectors are computed.

sense

CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If *sense* = 'N', none are computed;

If *sense* = 'E', computed for eigenvalues only;

If *sense* = 'V', computed for eigenvectors only;

If *sense* = 'B', computed for eigenvalues and eigenvectors.

n

INTEGER. The order of the matrices *A*, *B*, *vl*, and *vr* (*n*≥ 0).

a, *b*, *work*

REAL for sggevx

DOUBLE PRECISION for dggevx

COMPLEX for cggevx

DOUBLE COMPLEX for zggevx.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of the array *a*.

Must be at least max(1, *n*).

ldb

INTEGER. The leading dimension of the array *b*.

Must be at least max(1, *n*).

ldvl, *ldvr*

INTEGER. The leading dimensions of the output matrices *vl* and *vr*, respectively.

Constraints:

ldvl≥ 1. If *jobvl* = 'V', *ldvl*≥ max(1, *n*).

ldvr≥ 1. If *jobvr* = 'V', *ldvr*≥ max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*. *lwork*≥ max(1, 2*n*);

For real flavors:

If *balanc* = 'S', or 'B', or *jobvl* = 'V', or *jobvr* = 'V', then *lwork*≥ max(1, 6*n*);

if *sense* = 'E', or 'B', then *lwork*≥ max(1, 10*n*);

if *sense* = 'V', or 'B', *lwork*≥ $(2n^2+ 8*n+16)$.

For complex flavors:

if *sense* = 'E', *lwork*≥ max(1, 4*n*);

if *sense* = 'V', or 'B', *lwork*≥max(1, $2*n^2+ 2*n$).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork

REAL for cggevx

DOUBLE PRECISION for zggevx

Workspace array, size at least max(1, 6*n) if *balanc* = 'S', or 'B', and at least max(1, 2*n) otherwise.

This array is used in complex flavors only.

iwork

INTEGER.

Workspace array, size at least (*n*+6) for real flavors and at least (*n*+2) for complex flavors.

Not referenced if *sense* = 'E'.

bwork

LOGICAL. Workspace array, size at least max(1, *n*).

Not referenced if *sense* = 'N'.

Output Parameters

a, *b*

On exit, these arrays have been overwritten.

If *jobvl* = 'V' or *jobvr* = 'V' or both, then *a* contains the first part of the real Schur form of the "balanced" versions of the input *A* and *B*, and *b* contains its second part.

alphar, *alphai*

REAL for sggevx;

DOUBLE PRECISION for dggevx.

Arrays, size at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha

COMPLEX for cggevx;

DOUBLE COMPLEX for zggevx.

Array, size at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

REAL for sggevx

DOUBLE PRECISION for dggevx

COMPLEX for cggevx

DOUBLE COMPLEX for zggevx.

Array, size at least max(1, *n*).

For real flavors:

On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,..., *n*, will be the generalized eigenvalues.

If *alphai*(j) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.

For complex flavors:

On exit, *alpha*(j)/*beta*(j), j=1,..., *n*, will be the generalized eigenvalues.

See also *Application Notes* below.

vl, *vr*

REAL for sggevx

DOUBLE PRECISION for dggevx

COMPLEX for cggevx

DOUBLE COMPLEX for zggevx.

Arrays:

vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*).

If *jobvl* = 'V', the left generalized eigenvectors u(j) are stored one after another in the columns of *vl*, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) = 1.

If *jobvl* = 'N', *vl* is not referenced.

For real flavors:

If the j-th eigenvalue is real, then u(j) = *vl*(:,j), the j-th column of *vl*.

If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then for *i* = sqrt(–1), u(j) = *vl*(:,j) + *i***vl*(:,j+1) and u(j+1) = *vl*(:,j) – *i***vl*(:,j+1).

For complex flavors:

u(j) = *vl*(:,j), the j-th column of *vl*.

vr(*ldvr*,*); the second dimension of *vr* must be at least max(1, *n*).

If *jobvr* = 'V', the right generalized eigenvectors *v*(j) are stored one after another in the columns of *vr*, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) = 1.

If *jobvr* = 'N', *vr* is not referenced.

For real flavors:

If the j-th eigenvalue is real, then v(j) = *vr*(:,j), the j-th column of *vr*.

If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then v(j) = *vr*(:,j) + *i***vr*(:,j+1) and v(j+1) = *vr*(:,j) – *i***vr*(:,j+1).

For complex flavors:

v(j) = *vr*(:,j), the j-th column of *vr*.

ilo, *ihi*

INTEGER. *ilo* and *ihi* are integer values such that on exit $A_{i\ j} = 0$ and $B_{i\ j} = 0$ if i > j and j = 1,..., *ilo*–1 or i = *ihi*+1,..., *n*.

If *balanc* = 'N' or 'S', *ilo* = 1 and *ihi* = *n*.

lscale, *rscale*

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays, size at least max(1, *n*) each.

lscale contains details of the permutations and scaling factors applied to the left side of *A* and *B*.

If *PL*(j) is the index of the row interchanged with row j, and *DL*(j) is the scaling factor applied to row j, then

lscale(j) = *PL*(j), for j = 1,..., *ilo*-1

= *DL*(j), for j = *ilo*,...,*ihi*

= *PL*(j) for j = *ihi*+1,..., *n*.

The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

rscale contains details of the permutations and scaling factors applied to the right side of *A* and *B*.

If *PR*(j) is the index of the column interchanged with column j, and *DR*(j) is the scaling factor applied to column j, then

rscale(j) = *PR*(j), for j = 1,..., *ilo*-1

= *DR*(j), for j = *ilo*,...,*ihi*

= *PR*(j) for j = *ihi*+1,..., *n*.

The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

abnrm, *bbnrm*

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

The one-norms of the balanced matrices *A* and *B*, respectively.

rconde, *rcondv*

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Arrays, size at least max(1, *n*) each.

If *sense* = 'E', or 'B', *rconde* contains the reciprocal condition numbers of the eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of *rconde* are set to the same value. Thus *rconde*(j), *rcondv*(j), and the *j*-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the *j*-th eigenpair, unless all eigenpairs are selected).

If *sense* = 'N', or 'V', *rconde* is not referenced.

If *sense* = 'V', or 'B', *rcondv* contains the estimated reciprocal condition numbers of the eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of *rcondv* are set to the same value.

If the eigenvalues cannot be reordered to compute *rcondv*(j)*rconde*[*j*], *rcondv*(j) is set to 0; this can only occur when the true value would be very small anyway.

If *sense* = 'N', or 'E', *rcondv* is not referenced.

work(1)

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and

i≤*n*:

the *QZ* iteration failed. No eigenvectors have been calculated, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info*+1,..., *n* should be correct.

i > *n*: errors that usually indicate LAPACK problems:

i = *n*+1: other than *QZ* iteration failed in hgeqz;

i = *n*+2: error return from tgevc.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ggevx interface are the following:

a Holds the matrix *A* of size (*n*, *n*).

b Holds the matrix *B* of size (*n*, *n*).

alphar Holds the vector of length *n*. Used in real flavors only.

alphai Holds the vector of length *n*. Used in real flavors only.

alpha Holds the vector of length *n*. Used in complex flavors only.

beta Holds the vector of length *n*.

vl Holds the matrix *VL* of size (*n*, *n*).

vr Holds the matrix *VR* of size (*n*, *n*).

lscale Holds the vector of length *n*.

rscale Holds the vector of length *n*.

rconde Holds the vector of length *n*.

rcondv Holds the vector of length *n*.

balanc Must be 'N', 'B', or 'P'. The default value is 'N'.

jobvl Restored based on the presence of the argument *vl* as follows:

jobvl = 'V', if *vl* is present,

jobvl = 'N', if *vl* is omitted.

jobvr Restored based on the presence of the argument *vr* as follows:

jobvr = 'V', if *vr* is present,

jobvr = 'N', if *vr* is omitted.

sense Restored based on the presence of arguments *rconde* and rcondv as follows:

sense = 'B', if both *rconde* and *rcondv* are present,

sense = 'E', if *rconde* is present and *rcondv* omitted,

sense = 'V', if *rconde* is omitted and *rcondv* present,

sense = 'N', if both *rconde* and *rcondv* are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?ggev3

Computes the generalized eigenvalues and the left and right generalized eigenvectors for a pair of matrices.

Syntax

```
call sggev3 (jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta, vl, ldvl, vr, ldvr, work, lwork, info )
```

```
call dggev3 (jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta, vl, ldvl, vr, ldvr, work, lwork, info )
```

```
call cggev3 (jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work, lwork, rwork, info )
```

```
call zggev3 (jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work, lwork, rwork, info )
```

Include Files

- mkl.fi

Description

For a pair of *n*-by-*n* real or complex nonsymmetric matrices (*A*, *B*), ?ggev3 computes the generalized eigenvalues, and optionally, the left and right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (*A*, *B*) is a scalar λ or a ratio *alpha*/*beta* = λ, such that *A* - λ*B* is singular. It is usually represented as the pair (*alpha*,*beta*), as there is a reasonable interpretation for *beta*=0, and even for both being zero.

For real flavors:

The right eigenvector v_j corresponding to the eigenvalue λ_j of (*A*, *B*) satisfies

$A * v_j = \lambda_j * B * v_j$.

The left eigenvector u_j corresponding to the eigenvalue λ_j of (*A*, *B*) satisfies

$u_j^H * A = \lambda_j * u_j^H * B$

where u_j^H is the conjugate-transpose of u_j.

For complex flavors:

The right generalized eigenvector v_j corresponding to the generalized eigenvalue λ_j of (A, B) satisfies

$A * v_j = \lambda_j * B * v_j$.

The left generalized eigenvector u_j corresponding to the generalized eigenvalues λ_j of (A, B) satisfies

$u_j^H * A = \lambda_j * u_j^H * B$

where u_j^H is the conjugate-transpose of u_j.

Input Parameters

jobvl	CHARACTER*1. = 'N': do not compute the left generalized eigenvectors; = 'V': compute the left generalized eigenvectors.
jobvr	CHARACTER*1. = 'N': do not compute the right generalized eigenvectors; = 'V': compute the right generalized eigenvectors.
n	INTEGER. The order of the matrices *A*, *B*, *VL*, and *VR*. *n*≥ 0.
a	REAL for sggev3 DOUBLE PRECISION for dggev3 COMPLEX for cggev3 DOUBLE COMPLEX for zggev3 Array, size (*lda*, *n*). On entry, the matrix *A* in the pair (*A*, *B*).
lda	INTEGER. The leading dimension of *a*. *lda*≥ max(1,*n*).
b	REAL for sggev3 DOUBLE PRECISION for dggev3 COMPLEX for cggev3 DOUBLE COMPLEX for zggev3 Array, size (*ldb*, *n*). On entry, the matrix *B* in the pair (*A*, *B*).
ldb	INTEGER. The leading dimension of *b*. *ldb*≥ max(1,*n*).
ldvl	INTEGER. The leading dimension of the matrix *VL*. *ldvl*≥ 1, and if *jobvl* = 'V', *ldvl*≥*n*.
ldvr	INTEGER. The leading dimension of the matrix *VR*. *ldvr*≥ 1, and if *jobvr* = 'V', *ldvr*≥*n*.

work

REAL for sggev3

DOUBLE PRECISION for dggev3

COMPLEX for cggev3

DOUBLE COMPLEX for zggev3

Array, size (MAX(1,*lwork*))

On exit, if *info* = 0, *work*(1) returns the optimal lwork.

lwork

INTEGER. The dimension of the array *work*.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal (*A*, *B*) of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork

REAL for cggev3

DOUBLE PRECISION for zggev3

Array, size (8*n).

Output Parameters

a

On exit, *a* is overwritten.

b

On exit, *b* is overwritten.

alphar

REAL for sggev3

DOUBLE PRECISION for dggev3

Array, size (n).

alphai

REAL for sggev3

DOUBLE PRECISION for dggev3

Array, size (n).

alpha

COMPLEX for cggev3

DOUBLE COMPLEX for zggev3

Array, size (n).

beta

REAL for sggev3

DOUBLE PRECISION for dggev3

COMPLEX for cggev3

DOUBLE COMPLEX for zggev3

Array, size (n).

For real flavors:

On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,...,n, are the generalized eigenvalues. If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j + 1) negative.

Note: the quotients *alphar*(*j*)/*beta*(*j*) and *alphai*(*j*)/*beta*(*j*) can easily over- or underflow, and *beta*(*j*) might even be zero. Thus, you should avoid computing the ratio *alpha*/*beta* by simply dividing *alpha* by *beta*. However, *alphar* and *alphai* are always less than and usually comparable with norm(*A*) in magnitude, and *beta* is always less than and usually comparable with norm(*B*).

For complex flavors:

On exit, *alpha*(*j*)/*beta*(*j*), *j*=1,...,*n*, are the generalized eigenvalues.

Note: the quotients *alpha*(*j*)/*beta*(*j*) may easily over- or underflow, and *beta*(*j*) can even be zero. Thus, you should avoid computing the ratio *alpha*/*beta* by simply dividing *alpha* by *beta*. However, *alpha* is always less than and usually comparable with norm(*A*) in magnitude, and *beta*is always less than and usually comparable with norm(*B*).

vl

REAL for sggev3

DOUBLE PRECISION for dggev3

COMPLEX for cggev3

DOUBLE COMPLEX for zggev3

Array, size (*ldvl*, *n*).

For real flavors:

If *jobvl* = 'V', the left eigenvectors u_j are stored one after another in the columns of *vl*, in the same order as their eigenvalues. If the *j*-th eigenvalue is real, then u_j = *vl*(:,*j*), the *j*-th column of *vl*. If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then u_j = *vl*(:,*j*)+i**vl*(:,*j*+1) and u_{j+1} = *vl*(:,*j*)-i**vl*(:,*j*+1).

Each eigenvector is scaled so the largest component has abs(real part)+abs(imag. part)=1.

Not referenced if *jobvl* = 'N'.

For complex flavors:

If *jobvl* = 'V', the left generalized eigenvectors u_j are stored one after another in the columns of *vl*, in the same order as their eigenvalues.

Each eigenvector is scaled so the largest component has abs(real part) + abs(imag. part) = 1.

Not referenced if *jobvl* = 'N'.

vr

REAL for sggev3

DOUBLE PRECISION for dggev3

COMPLEX for cggev3

DOUBLE COMPLEX for zggev3

Array, size (*ldvr*, *n*).

For real flavors:

If *jobvr* = 'V', the right eigenvectors v_j are stored one after another in the columns of *vr*, in the same order as their eigenvalues. If the *j*-th eigenvalue is real, then v_j = *vr*(:,*j*), the *j*-th column of *vr*. If the *j*-th and (*j* + 1)-st eigenvalues form a complex conjugate pair, then v_j = *vr*(:,*j*) + i**vr*(:,*j* + 1) and v_{j+1} = *vr*(:,*j*)-i**vr*(:,*j*+1).

Each eigenvector is scaled so the largest component has abs(real part)+abs(imag. part)=1.

Not referenced if *jobvr* = 'N'.

For complex flavors:

If *jobvr* = 'V', the right generalized eigenvectors v_j are stored one after another in the columns of *vr*, in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has abs(real part) + abs(imag. part) = 1.

Not referenced if *jobvr* = 'N'.

info

INTEGER. = 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

=1,...,*n*:

- for real flavors:

  The QZ iteration failed. No eigenvectors have been calculated, but *alphar*(*j*), *alphar*(*j*) and *beta*(*j*) should be correct for *j*=*info* + 1,...,*n*.
- for complex flavors:

  The QZ iteration failed. No eigenvectors have been calculated, but *alpha*(*j*) and *beta*(*j*) should be correct for *j*=*info* + 1,...,*n*.

> *n*:

- =*n* + 1: other than QZ iteration failed in ?hgeqz,
- =*n* + 2: error return from ?tgevc.

LAPACK Auxiliary Routines

Routine naming conventions, mathematical notation, and matrix storage schemes used for LAPACK auxiliary routines are the same as for the driver and computational routines described in previous chapters.

The table below summarizes information about the available LAPACK auxiliary routines.

LAPACK Auxiliary Routines

Routine Name	Data Types	Description
?lacgv	c, z	Conjugates a complex vector.
?lacrm	c, z	Multiplies a complex matrix by a square real matrix.
?lacrt	c, z	Performs a linear transformation of a pair of complex vectors.
?laesy	c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix.

Routine Name	Data Types	Description
?rot	c, z	Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.
?spmv	c, z	Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix
?spr	c, z	Performs the symmetrical rank-1 update of a complex symmetric packed matrix.
?syconv	s, c, d, z	Converts a symmetric matrix given by a triangular matrix factorization into two matrices and vice versa.
?symv	c, z	Computes a matrix-vector product for a complex symmetric matrix.
?syr	c, z	Performs the symmetric rank-1 update of a complex symmetric matrix.
i?max1	c, z	Finds the index of the vector element whose real part has maximum absolute value.
?sum1	sc, dz	Forms the 1-norm of the complex vector using the true absolute value.
?gbtf2	s, d, c, z	Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.
?gebd2	s, d, c, z	Reduces a general matrix to bidiagonal form using an unblocked algorithm.
?gehd2	s, d, c, z	Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.
?gelq2	s, d, c, z	Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.
?gelqt3	s, d, c, z	Recursively computes the LQ factorization of a general matrix using the compact WY representation of Q.
?geql2	s, d, c, z	Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.
?geqr2	s, d, c, z	Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.
?geqr2p	s, d, c, z	Computes the QR factorization of a general rectangular matrix with non-negative diagonal elements using an unblocked algorithm.
?geqrt2	s, d, c, z	Computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.
?geqrt3	s, d, c, z	Recursively computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.
?gerq2	s, d, c, z	Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.
?gesc2	s, d, c, z	Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2.

Routine Name	Data Types	Description
?getc2	s, d, c, z	Computes the LU factorization with complete pivoting of the general *n*-by-*n* matrix.
?getf2	s, d, c, z	Computes the LU factorization of a general *m*-by-*n* matrix using partial pivoting with row interchanges (unblocked algorithm).
?gtts2	s, d, c, z	Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.
?isnan	s, d,	Tests input for NaN.
?laisnan	s, d,	Tests input for NaN by comparing two arguments for inequality.
?labrd	s, d, c, z	Reduces the first *nb* rows and columns of a general matrix to a bidiagonal form.
?lacn2	s, d, c, z	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacon	s, d, c, z	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacpy	s, d, c, z	Copies all or part of one two-dimensional array to another.
?ladiv	s, d, c, z	Performs complex division in real arithmetic, avoiding unnecessary overflow.
?lae2	s, d	Computes the eigenvalues of a 2-by-2 symmetric matrix.
?laebz	s, d	Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine ?stebz.
?laed0	s, d, c, z	Used by ?stedc. Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.
?laed1	s, d	Used by sstedc/dstedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.
?laed2	s, d	Used by sstedc/dstedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.
?laed3	s, d	Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.
?laed4	s, d	Used by sstedc/dstedc. Finds a single root of the secular equation.
?laed5	s, d	Used by sstedc/dstedc. Solves the 2-by-2 secular equation.
?laed6	s, d	Used by sstedc/dstedc. Computes one Newton step in solution of the secular equation.
?laed7	s, d, c, z	Used by ?stedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.

Routine Name	Data Types	Description
?laed8	s, d, c, z	Used by ?stedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.
?laed9	s, d	Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.
?laeda	s, d	Used by ?stedc. Computes the *Z* vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.
?laein	s, d, c, z	Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.
?laev2	s, d, c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.
?laexc	s, d	Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
?lag2	s, d	Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.
?lags2	s, d	Computes 2-by-2 orthogonal matrices *U*, *V*, and *Q*, and applies them to matrices *A* and *B* such that the rows of the transformed *A* and *B* are parallel.
?lagtf	s, d	Computes an LU factorization of a matrix $T-\lambda I$, where *T* is a general tridiagonal matrix, and λ a scalar, using partial pivoting with row interchanges.
?lagtm	s, d, c, z	Performs a matrix-matrix product of the form $C = \alpha AB+\beta C$, where *A* is a tridiagonal matrix, *B* and *C* are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.
?lagts	s, d	Solves the system of equations $(T-\lambda I)x = y$ or $(T-\lambda I)^T x = y$,where *T* is a general tridiagonal matrix and λ a scalar, using the LU factorization computed by ?lagtf.
?lagv2	s, d	Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (*A*,*B*) where *B* is upper triangular.
?lahqr	s, d, c, z	Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.
?lahrd	s, d, c, z	Reduces the first *nb* columns of a general rectangular matrix *A* so that elements below the *k*-th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of *A*.
?lahr2	s, d, c, z	Reduces the specified number of first columns of a general rectangular matrix *A* so that elements below the specified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of *A*.
?laic1	s, d, c, z	Applies one step of incremental condition estimation.

Routine Name	Data Types	Description
?lakf2	s, d, c, z	Forms a matrix containing Kronecker products between the given matrices.
?lals0	s, d, c, z	Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd.
?lalsa	s, d, c, z	Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.
?lalsd	s, d, c, z	Uses the singular value decomposition of *A* to solve the least squares problem.
?lamrg	s, d	Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in ascending order.
?lamswlq	s, d, c, z	Multiplies a general real matrix by a real orthogonal matrix defined as the product of blocked elementary reflectors computed by short wide LQ factorization.
?lamtsqr	s, d, c, z	Multiplies a general matrix by the product of blocked elementary reflectors computed by tall skinny QR factorization.
?laneg	s, d	Computes the Sturm count.
?langb	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.
?lange	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.
?langt	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.
?lanhs	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.
?lansb	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.
?lanhb	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.
?lansp	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.
?lanhp	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.
?lanst/?lanht	s, d/c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.

Routine Name	Data Types	Description
?lansy	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/ complex symmetric matrix.
?lanhe	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.
?lantb	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.
?lantp	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.
?lantr	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.
?lanv2	s, d	Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.
?lapll	s, d, c, z	Measures the linear dependence of two vectors.
?lapmr	s, d, c, z	Rearranges rows of a matrix as specified by a permutation vector.
?lapmt	s, d, c, z	Performs a forward or backward permutation of the columns of a matrix.
?lapy2	s, d	Returns sqrt(x^2+y^2).
?lapy3	s, d	Returns sqrt($x^2+y^2+z^2$).
?laqgb	s, d, c, z	Scales a general band matrix, using row and column scaling factors computed by ?gbequ.
?laqge	s, d, c, z	Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.
?laqhb	c, z	Scales a Hermitian band matrix, using scaling factors computed by ?pbequ.
?laqp2	s, d, c, z	Computes a QR factorization with column pivoting of the matrix block.
?laqps	s, d, c, z	Computes a step of QR factorization with column pivoting of a real m-by-n matrix *A* by using BLAS level 3.
?laqr0	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.
?laqr1	s, d, c, z	Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix *H* and specified shifts.
?laqr2	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Routine Name	Data Types	Description
?laqr3	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
?laqr4	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.
?laqr5	s, d, c, z	Performs a single small-bulge multi-shift QR sweep.
?laqsb	s, d, c, z	Scales a symmetric/Hermitian band matrix, using scaling factors computed by ?pbequ.
?laqsp	s, d, c, z	Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.
?laqsy	s, d, c, z	Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.
?laqtr	s, d	Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.
?lar1v	s, d, c, z	Computes the (scaled) *r*-th column of the inverse of the submatrix in rows *b1* through *bn* of the tridiagonal matrix $LDL^T - \lambda I$.
?lar2v	s, d, c, z	Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.
?laran	s, d	Returns a random real number from a uniform distribution.
?larf	s, d, c, z	Applies an elementary reflector to a general rectangular matrix.
?larfb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
?larfg	s, d, c, z	Generates an elementary reflector (Householder matrix).
?larfgp	s, d, c, z	Generates an elementary reflector (Householder matrix) with non-negatibe beta.
?larft	s, d, c, z	Forms the triangular factor *T* of a block reflector $H = I - vtv^H$
?larfx	s, d, c, z	Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order ≤ 10.
?large	s, d, c, z	Pre- and post-multiplies a real general matrix with a random orthogonal matrix.
?larnd	s, d, c, z	Returns a random real number from a uniform or normal distribution.
?largv	s, d, c, z	Generates a vector of plane rotations with real cosines and real/complex sines.
?larnv	s, d, c, z	Returns a vector of random numbers from a uniform or normal distribution.
?laror	s, d, c, z	Pre- or post-multiplies an *m*-by-*n* matrix by a random orthogonal/unitary matrix.

Routine Name	Data Types	Description
?larot	s, d, c, z	Applies a Givens rotation to two adjacent rows or columns.
?larra	s, d	Computes the splitting points with the specified threshold.
?larrb	s, d	Provides limited bisection to locate eigenvalues for more accuracy.
?larrc	s, d	Computes the number of eigenvalues of the symmetric tridiagonal matrix.
?larrd	s, d	Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.
?larre	s, d	Given the tridiagonal matrix T, sets small off-diagonal elements to zero and for each unreduced block T_i, finds base representations and eigenvalues.
?larrf	s, d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrj	s, d	Performs refinement of the initial estimates of the eigenvalues of the matrix T.
?larrk	s, d	Computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy.
?larrr	s, d	Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.
?larrv	s, d, c, z	Computes the eigenvectors of the tridiagonal matrix $T = LDL^T$ given L, D and the eigenvalues of LDL^T.
?lartg	s, d, c, z	Generates a plane rotation with real cosine and real/complex sine.
?lartgp	s, d	Generates a plane rotation so that the diagonal is nonnegative.
?lartgs	s, d	Generates a plane rotation designed to introduce a bulge in implicit QR iteration for the bidiagonal SVD problem.
?lartv	s, d, c, z	Applies a vector of plane rotations with real cosines and real/ complex sines to the elements of a pair of vectors.
?laruv	s, d	Returns a vector of n random real numbers from a uniform distribution.
?larz	s, d, c, z	Applies an elementary reflector (as returned by ?tzrzf) to a general matrix.
?larzb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general matrix.
?larzt	s, d, c, z	Forms the triangular factor T of a block reflector $H = I - vtv^H$.
?las2	s, d	Computes singular values of a 2-by-2 triangular matrix.
?lascl	s, d, c, z	Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from}.
?lasd0	s, d	Computes the singular values of a real upper bidiagonal n-by-m matrix B with diagonal d and off-diagonal e. Used by ?bdsdc.

Routine Name	Data Types	Description
?lasd1	s, d	Computes the SVD of an upper bidiagonal matrix *B* of the specified size. Used by ?bdsdc.
?lasd2	s, d	Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.
?lasd3	s, d	Finds all square roots of the roots of the secular equation, as defined by the values in *D* and *Z*, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.
?lasd4	s, d	Computes the square root of the *i*-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc.
?lasd5	s, d	Computes the square root of the *i*-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by ?bdsdc.
?lasd6	s, d	Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.
?lasd7	s, d	Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.
?lasd8	s, d	Finds the square roots of the roots of the secular equation, and stores, for each element in *D*, the distance to its two nearest poles. Used by ?bdsdc.
?lasda	s, d	Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal *d* and off-diagonal *e*. Used by ?bdsdc.
?lasdq	s, d	Computes the SVD of a real bidiagonal matrix with diagonal *d* and off-diagonal *e*. Used by ?bdsdc.
?lasdt	s, d	Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.
?laset	s, d, c, z	Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.
?lasq1	s, d	Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.
?lasq2	s, d	Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the *qd* Array *Z* to high relative accuracy. Used by ?bdsqr and ?stegr.
?lasq3	s, d	Checks for deflation, computes a shift and calls *dqds*. Used by ?bdsqr.
?lasq4	s, d	Computes an approximation to the smallest eigenvalue using values of *d* from the previous transform. Used by ?bdsqr.
?lasq5	s, d	Computes one *dqds* transform in ping-pong form. Used by ?bdsqr and ?stegr.

Routine Name	Data Types	Description
?lasq6	s, d	Computes one *dqd* transform in ping-pong form. Used by ?bdsqr and ?stegr.
?lasr	s, d, c, z	Applies a sequence of plane rotations to a general rectangular matrix.
?lasrt	s, d	Sorts numbers in increasing or decreasing order.
?lassq	s, d, c, z	Updates a sum of squares represented in scaled form.
?lasv2	s, d	Computes the singular value decomposition of a 2-by-2 triangular matrix.
?laswp	s, d, c, z	Performs a series of row interchanges on a general rectangular matrix.
?laswlq	s, d, c, z	Computes blocked short-wide LQ matrix factorization.
?lasy2	s, d	Solves the Sylvester matrix equation where the matrices are of order 1 or 2.
?lasyf	s, d, c, z	Computes a partial factorization of a real/complex symmetric matrix, using the diagonal pivoting method.
?lasyf_aa	s, d, c, z	Factorizes a panel of a symmetric matrix using Aasen's algorithm.
?lasyf_rook	s, d, c, z	Computes a partial factorization of a real/complex symmetric matrix, using the bounded Bunch-Kaufman diagonal pivoting method.
?lahef	c, z	Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.
?lahef_rook	c, z	Computes a partial factorization of a complex Hermitian indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method.
?lahef_aa	c, z	Computes a partial factorization of a complex Hermitian matrix, using Aasen's algorithm.
?latbs	s, d, c, z	Solves a triangular banded system of equations.
?latdf	s, d, c, z	Uses the LU factorization of the *n*-by-*n* matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.
?latm1	s, d, c, z	Computes the entries of a matrix as specified.
?latm2	s, d, c, z	Returns an entry of a random matrix.
?latm3	s, d, c, z	Returns set entry of a random matrix.
?latm5	s, d, c, z	Generates matrices involved in the Generalized Sylvester equation.
?latm6	s, d, c, z	Generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.

Routine Name	Data Types	Description
?latme	s, d, c, z	Generates random non-symmetric square matrices with specified eigenvalues.
?latmr	s, d, c, z	Generates random matrices of various types.
?latps	s, d, c, z	Solves a triangular system of equations with the matrix held in packed storage.
?latrd	s, d, c, z	Reduces the first *nb* rows and columns of a symmetric/Hermitian matrix *A* to real tridiagonal form by an orthogonal/unitary similarity transformation.
?latrs	s, d, c, z	Solves a triangular system of equations with the scale factor set to prevent overflow.
?latrz	s, d, c, z	Factors an upper trapezoidal matrix by means of orthogonal/ unitary transformations.
?latsqr	s, d, c, z	Computes a blocked tall-skinny QR matrix factorization.
?lauu2	s, d, c, z	Computes the product UU^H or L^HL, where *U* and *L* are upper or lower triangular matrices (unblocked algorithm).
?lauum	s, d, c, z	Computes the product UU^H or L^HL, where *U* and *L* are upper or lower triangular matrices (blocked algorithm).
?orbdb1/?unbdb1 ?orbdb2/?unbdb2 ?orbdb3/?unbdb3 ?orbdb4/?unbdb4	s, d, c, z	Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.
?orbdb5/?unbdb5 ?orbdb6/?unbdb6	s, d, c, z	Orthogonalizes a column vector with respect to the orthonormal basis matrix.
?org2l/?ung2l	s, d/c, z	Generates all or part of the orthogonal/unitary matrix *Q* from a QL factorization determined by ?geqlf (unblocked algorithm).
?org2r/?ung2r	s, d/c, z	Generates all or part of the orthogonal/unitary matrix *Q* from a QR factorization determined by ?geqrf (unblocked algorithm).
?orgl2/?ungl2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix *Q* from an LQ factorization determined by ?gelqf (unblocked algorithm).
?orgr2/?ungr2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix *Q* from an RQ factorization determined by ?gerqf (unblocked algorithm).
?orm2l/?unm2l	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf (unblocked algorithm).
?orm2r/?unm2r	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf (unblocked algorithm).
?orml2/?unml2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf (unblocked algorithm).
?ormr2/?unmr2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf (unblocked algorithm).

Routine Name	Data Types	Description
?ormr3/?unmr3	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzrzf (unblocked algorithm).
?pbtf2	s, d, c, z	Computes the Cholesky factorization of a symmetric/ Hermitian positive definite band matrix (unblocked algorithm).
?potf2	s, d, c, z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (unblocked algorithm).
?ptts2	s, d, c, z	Solves a tridiagonal system of the form $AX=B$ using the LDL^H factorization computed by ?pttrf.
?rscl	s, d, cs, zd	Multiplies a vector by the reciprocal of a real scalar.
?syswapr	s, d, c, z	Applies an elementary permutation on the rows and columns of a symmetric matrix.
?heswapr	c, z	Applies an elementary permutation on the rows and columns of a Hermitian matrix.
?sygs2/?hegs2	s, d/c, z	Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf (unblocked algorithm).
?sytd2/?hetd2	s, d/c, z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (unblocked algorithm).
?sytf2	s, d, c, z	Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).
?sytf2_rook	s, d, c, z	Computes the factorization of a real/complex symmetric indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).
?hetf2	c, z	Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).
?hetf2_rook	c, z	Computes the factorization of a complex Hermitian matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).
?tgex2	s, d, c, z	Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.
?tgsy2	s, d, c, z	Solves the generalized Sylvester equation (unblocked algorithm).
?trti2	s, d, c, z	Computes the inverse of a triangular matrix (unblocked algorithm).
clag2z	c → z	Converts a complex single precision matrix to a complex double precision matrix.
dlag2s	d → s	Converts a double precision matrix to a single precision matrix.
slag2d	s → d	Converts a single precision matrix to a double precision matrix.
zlag2c	z → c	Converts a complex double precision matrix to a complex single precision matrix.

Routine Name	Data Types	Description
?larfp	s, d, c, z	Generates a real or complex elementary reflector.
ila?lc	s, d, c, z	Scans a matrix for its last non-zero column.
ila?lr	s, d, c, z	Scans a matrix for its last non-zero row.
?gsvj0	s, d	Pre-processor for the routine ?gesvj.
?gsvj1	s, d	Pre-processor for the routine ?gesvj, applies Jacobi rotations targeting only particular pivots.
?sfrk	s, d	Performs a symmetric rank-k operation for matrix in RFP format.
?hfrk	c, z	Performs a Hermitian rank-k operation for matrix in RFP format.
?tfsm	s, d, c, z	Solves a matrix equation (one operand is a triangular matrix in RFP format).
?lansf	s, d	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix in RFP format.
?lanhf	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian matrix in RFP format.
?tfttp	s, d, c, z	Copies a triangular matrix from the rectangular full packed format (TF) to the standard packed format (TP).
?tfttr	s, d, c, z	Copies a triangular matrix from the rectangular full packed format (TF) to the standard full format (TR).
?tplqt2	s, d, c, z	Computes an LQ factorization of a triangular-pentagonal matrix using the compact WY representation for Q.
?tpqrt2	s, d, c, z	Computes a QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q.
?tprfb	s, d, c, z	Applies a real or complex "triangular-pentagonal" blocked reflector to a real or complex matrix, which is composed of two blocks.
?tpttf	s, d, c, z	Copies a triangular matrix from the standard packed format (TP) to the rectangular full packed format (TF).
?tpttr	s, d, c, z	Copies a triangular matrix from the standard packed format (TP) to the standard full format (TR).
?trttf	s, d, c, z	Copies a triangular matrix from the standard full format (TR) to the rectangular full packed format (TF).
?trttp	s, d, c, z	Copies a triangular matrix from the standard full format (TR) to the standard packed format (TP).
?pstf2	s, d, c, z	Computes the Cholesky factorization with complete pivoting of a real symmetric or complex Hermitian positive semi-definite matrix.
dlat2s	d → s	Converts a double-precision triangular matrix to a single-precision triangular matrix.

Routine Name	Data Types	Description
zlat2c	z → c	Converts a double complex triangular matrix to a complex triangular matrix.
?lacp2	c, z	Copies all or part of a real two-dimensional array to a complex array.
?la_gbamv	s, d, c, z	Performs a matrix-vector operation to calculate error bounds.
?la_gbrcond	s, d	Estimates the Skeel condition number for a general banded matrix.
?la_gbrcond_c	c, z	Computes the infinity norm condition number of op(A)*inv(diag(c)) for general banded matrices.
?la_gbrcond_x	c, z	Computes the infinity norm condition number of op(A)*diag(x) for general banded matrices.
?la_gbrfsx_extended	s, d, c, z	Improves the computed solution to a system of linear equations for general banded matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_gbrpvgrw	s, d, c, z	Computes the reciprocal pivot growth factor norm(A)/norm(U) for a general banded matrix.
?la_geamv	s, d, c, z	Computes a matrix-vector product using a general matrix to calculate error bounds.
?la_gercond	s, d	Estimates the Skeel condition number for a general matrix.
?la_gercond_c	c, z	Computes the infinity norm condition number of op(A)*inv(diag(c)) for general matrices.
?la_gercond_x	c, z	Computes the infinity norm condition number of op(A)*diag(x) for general matrices.
?la_gerfsx_extended	s, d	Improves the computed solution to a system of linear equations for general matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_heamv	c, z	Computes a matrix-vector product using a Hermitian indefinite matrix to calculate error bounds.
?la_hercond_c	c, z	Computes the infinity norm condition number of op(A)*inv(diag(c)) for Hermitian indefinite matrices.
?la_hercond_x	c, z	Computes the infinity norm condition number of op(A)*diag(x) for Hermitian indefinite matrices.
?la_herfsx_extended	c, z	Improves the computed solution to a system of linear equations for Hermitian indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_lin_berr	s, d, c, z	Computes a component-wise relative backward error.
?la_porcond	s, d	Estimates the Skeel condition number for a symmetric positive-definite matrix.
?la_porcond_c	c, z	Computes the infinity norm condition number of op(A)*inv(diag(c)) for Hermitian positive-definite matrices.

Routine Name	Data Types	Description
?la_porcond_x	c, z	Computes the infinity norm condition number of op(A)*diag(x) for Hermitian positive-definite matrices.
?la_porfsx_extended	s, d, c, z	Improves the computed solution to a system of linear equations for symmetric or Hermitian positive-definite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_porpvgrw	s, d, c, z	Computes the reciprocal pivot growth factor norm(A)/norm(U) for a symmetric or Hermitian positive-definite matrix.
?laqhe	c, z	Scales a Hermitian matrix.
?laqhp	c, z	Scales a Hermitian matrix stored in packed form.
?larcm	c, z	Copies all or part of a real two-dimensional array to a complex array.
?la_rpvgrw	c, z	Multiplies a square real matrix by a complex matrix.
?larscl2	s, d, c, z	Performs reciprocal diagonal scaling on a vector.
?lascl2	s, d, c, z	Performs diagonal scaling on a vector.
?la_syamv	s, d, c, z	Computes a matrix-vector product using a symmetric indefinite matrix to calculate error bounds.
?la_syrcond	s, d	Estimates the Skeel condition number for a symmetric indefinite matrix.
?la_syrcond_c	c, z	Computes the infinity norm condition number of op(A)*inv(diag(c)) for symmetric indefinite matrices.
?la_syrcond_x	c, z	Computes the infinity norm condition number of op(A)*diag(x) for symmetric indefinite matrices.
?la_syrfsx_extended	s, d, c, z	Improves the computed solution to a system of linear equations for symmetric indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_syrpvgrw	s, d, c, z	Computes the reciprocal pivot growth factor norm(A)/norm(U) for a symmetric indefinite matrix.
?la_wwaddw	s, d, c, z	Adds a vector into a doubled-single vector.
?larfy	s, d, c, z	Applies an elementary reflector, or Householder matrix, H, to an *n* by *n* symmetric or Hermitian matrix C, from both the left and the right.
mkl_?tppack	s, d, c, z	Copies a triangular/symmetric matrix or submatrix from standard full format to standard packed format.
mkl_?tpunpack	s, d, c, z	Copies a triangular/symmetric matrix or submatrix from standard packed format to full format.

?lacgv

Conjugates a complex vector.

Syntax

```
call clacgv( n, x, incx )
call zlacgv( n, x, incx )
```

Include Files

- mkl.fi

Description

The routine conjugates a complex vector *x* of length *n* and increment *incx* (see "Vector Arguments in BLAS" in Appendix B).

Input Parameters

The data types are given for the Fortran interface.

n	INTEGER. The length of the vector *x* (*n*≥ 0).
x	COMPLEX for clacgv DOUBLE COMPLEX for zlacgv. Array, dimension (1+(*n*-1)* |*incx*|). Contains the vector of length *n* to be conjugated.
incx	INTEGER. The spacing between successive elements of *x*.

Output Parameters

x	On exit, overwritten with conjg(*x*).

?lacrm

Multiplies a complex matrix by a square real matrix.

Syntax

```
call clacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
call zlacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

Include Files

- mkl.fi

Description

The routine performs a simple matrix-matrix multiplication of the form

*C = A*B*,

where *A* is *m*-by-*n* and complex, *B* is *n*-by-*n* and real, *C* is *m*-by-*n* and complex.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* and of the matrix *C* (*m*≥ 0).

n	INTEGER. The number of columns and rows of the matrix *B* and the number of columns of the matrix *C* (*n*≥ 0).
a	COMPLEX for clacrm DOUBLE COMPLEX for zlacrm Array, DIMENSION(*lda*, *n*). Contains the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*, *lda*≥max(1, *m*).
b	REAL for clacrm DOUBLE PRECISION for zlacrm Array, DIMENSION(*ldb*, *n*). Contains the *n*-by-*n* matrix *B*.
ldb	INTEGER. The leading dimension of the array *b*, *ldb*≥max(1, *n*).
ldc	INTEGER. The leading dimension of the output array *c*, *ldc*≥max(1, *n*).
rwork	REAL for clacrm DOUBLE PRECISION for zlacrm Workspace array, DIMENSION(2*m*n).

Output Parameters

c	COMPLEX for clacrm DOUBLE COMPLEX for zlacrm Array, DIMENSION (*ldc*, *n*). Contains the *m*-by-*n* matrix *C*.

?lacrt

Performs a linear transformation of a pair of complex vectors.

Syntax

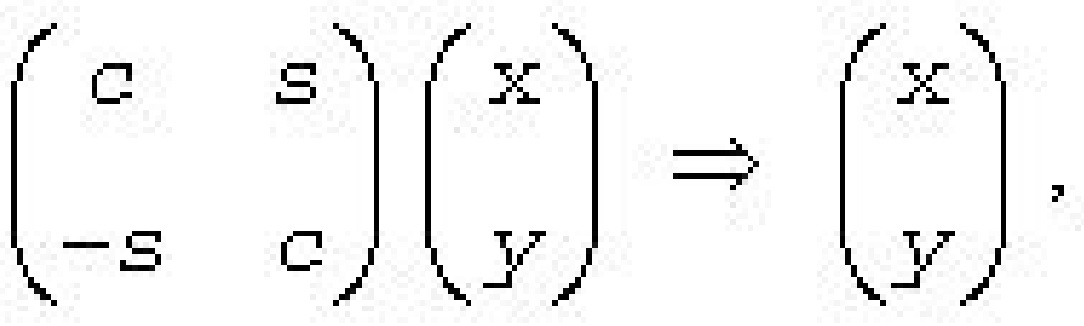
```
call clacrt( n, cx, incx, cy, incy, c, s )
call zlacrt( n, cx, incx, cy, incy, c, s )
```

Include Files

- mkl.fi

Description

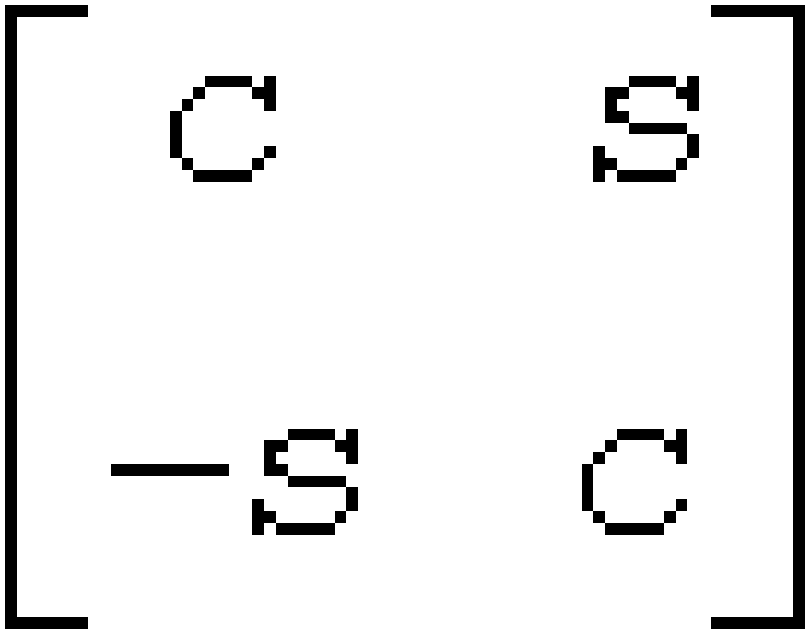
The routine performs the following transformation



where *c*, *s* are complex scalars and *x*, *y* are complex vectors.

Input Parameters

n	INTEGER. The number of elements in the vectors *cx* and *cy* (*n*≥ 0).
cx, *cy*	COMPLEX for clacrt DOUBLE COMPLEX for zlacrt Arrays, dimension (*n*). Contain input vectors *x* and *y*, respectively.
incx	INTEGER. The increment between successive elements of *cx*.
incy	INTEGER. The increment between successive elements of *cy*.
c, *s*	COMPLEX for clacrt DOUBLE COMPLEX for zlacrt Complex scalars that define the transform matrix



Output Parameters

cx	On exit, overwritten with *c*x + s*y*.
cy	On exit, overwritten with *-s*x + c*y*.

?laesy

Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix, and checks that the norm of the matrix of eigenvectors is larger than a threshold value.

Syntax

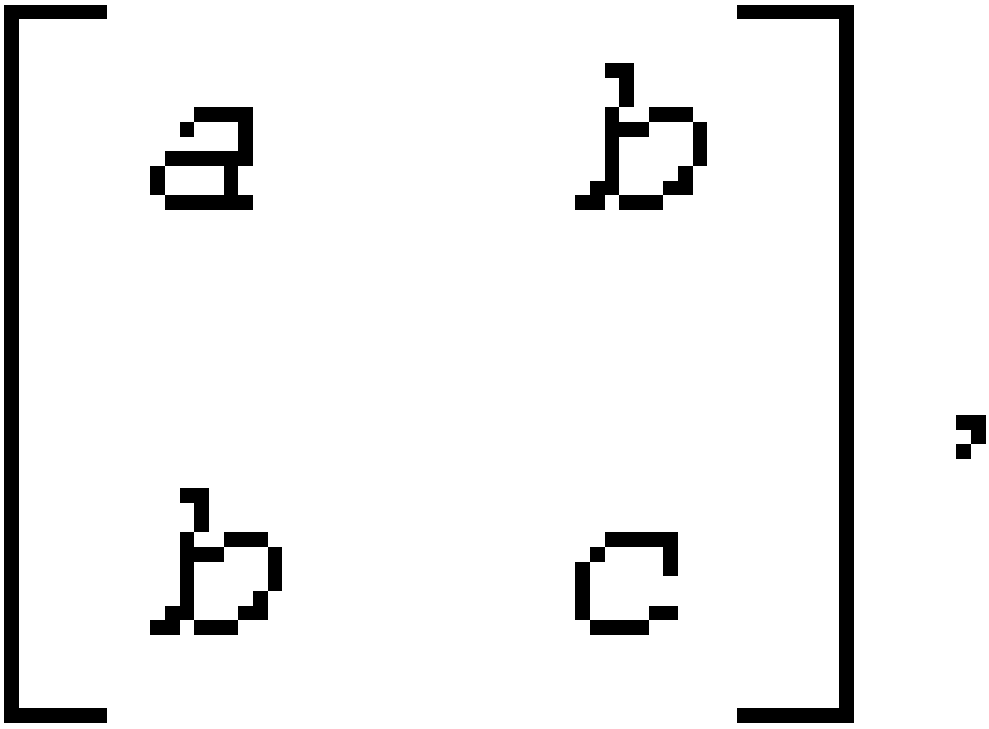
```
call claesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
call zlaesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
```

Include Files

- mkl.fi

Description

The routine performs the eigendecomposition of a 2-by-2 symmetric matrix



provided the norm of the matrix of eigenvectors is larger than some threshold value.

rt1 is the eigenvalue of larger absolute value, and *rt2* of smaller absolute value. If the eigenvectors are computed, then on return (*cs1*, *sn1*) is the unit eigenvector for *rt1*, hence

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -sn1 \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

Input Parameters

a, *b*, *c*

`COMPLEX` for `claesy`

`DOUBLE COMPLEX` for `zlaesy`

Elements of the input matrix.

Output Parameters

rt1, *rt2*

`COMPLEX` for `claesy`

`DOUBLE COMPLEX` for `zlaesy`

Eigenvalues of larger and smaller modulus, respectively.

evscal

`COMPLEX` for `claesy`

`DOUBLE COMPLEX` for `zlaesy`

The complex value by which the eigenvector matrix was scaled to make it orthonormal. If *evscal* is zero, the eigenvectors were not computed. This means one of two things: the 2-by-2 matrix could not be diagonalized, or the norm of the matrix of eigenvectors before scaling was larger than the threshold value `thresh` (set to 0.1E0).

cs1, *sn1*

`COMPLEX` for `claesy`

`DOUBLE COMPLEX` for `zlaesy`

If *evscal* is not zero, then (*cs1*, *sn1*) is the unit right eigenvector for *rt1*.

?rot

Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.

Syntax

```
call crot( n, cx, incx, cy, incy, c, s )
call zrot( n, cx, incx, cy, incy, c, s )
```

Include Files

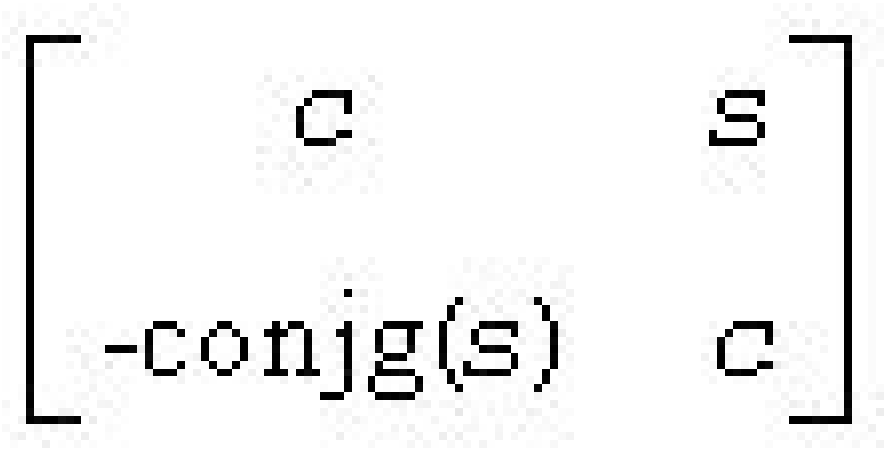
- `mkl.fi`

Description

The routine applies a plane rotation, where the cosine (*c*) is real and the sine (*s*) is complex, and the vectors *cx* and *cy* are complex. This routine has its real equivalents in BLAS (see `?rot` in Chapter "BLAS and Sparse BLAS Routines").

Input Parameters

n	INTEGER. The number of elements in the vectors *cx* and *cy*.
cx, *cy*	REAL for srot DOUBLE PRECISION for drot COMPLEX for crot DOUBLE COMPLEX for zrot Arrays of dimension (*n*), contain input vectors *x* and *y*, respectively.
incx	INTEGER. The increment between successive elements of *cx*.
incy	INTEGER. The increment between successive elements of *cy*.
c	REAL for crot DOUBLE PRECISION for zrot
s	REAL for srot DOUBLE PRECISION for drot COMPLEX for crot DOUBLE COMPLEX for zrot Values that define a rotation



where *c*c + s*conjg(s) = 1.0*.

Output Parameters

cx	On exit, overwritten with *c*x + s*y*.
cy	On exit, overwritten with -conjg(*s*)*x + *c*y*.

?spmv

Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix.

Syntax

call cspmv(*uplo*, *n*, *alpha*, *ap*, *x*, *incx*, *beta*, *y*, *incy*)

```
call zspmv( uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Include Files

- mkl.fi

Description

The ?spmv routines perform a matrix-vector operation defined as

y := *alpha***a***x* + *beta***y*,

where:

alpha and *beta* are complex scalars,

x and *y* are *n*-element complex vectors

a is an *n*-by-*n* complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see ?spmv in Chapter "BLAS and Sparse BLAS Routines").

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*.

If *uplo* = 'U' or 'u', the upper triangular part of the matrix *a* is supplied in the array *ap*.

If *uplo* = 'L' or 'l', the lower triangular part of the matrix *a* is supplied in the array *ap* .

n

INTEGER.

Specifies the order of the matrix *a*.

The value of *n* must be at least zero.

alpha, *beta*

COMPLEX for cspmv

DOUBLE COMPLEX for zspmv

Specify complex scalars *alpha* and *beta*. When *beta* is supplied as zero, then *y* need not be set on input.

ap

COMPLEX for cspmv

DOUBLE COMPLEX for zspmv

Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry, with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *A*(1, 1), *ap*(2) and *ap*(3) contain *A*(1, 2) and *A*(2, 2) respectively, and so on. Before entry, with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on.

x

COMPLEX for cspmv

DOUBLE COMPLEX for zspmv

	Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array x must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	COMPLEX for cspmv DOUBLE COMPLEX for zspmv Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

?spr

Performs the symmetrical rank-1 update of a complex symmetric packed matrix.

Syntax

```
call cspr( uplo, n, alpha, x, incx, ap )
call zspr( uplo, n, alpha, x, incx, ap )
```

Include Files

- mkl.fi

Description

The ?spr routines perform a matrix-vector operation defined as

$a := alpha*x*x^H + a$,

where:

alpha is a complex scalar

x is an *n*-element complex vector

a is an *n*-by-*n* complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see ?spr in Chapter "BLAS and Sparse BLAS Routines").

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows: If *uplo* = 'U' or 'u', the upper triangular part of the matrix *a* is supplied in the array *ap*. If *uplo* = 'L' or 'l', the lower triangular part of the matrix *a* is supplied in the array *ap* .

n INTEGER.

Specifies the order of the matrix *a*.

The value of *n* must be at least zero.

alpha COMPLEX for cspr

DOUBLE COMPLEX for zspr

Specifies the scalar *alpha*.

x COMPLEX for cspr

DOUBLE COMPLEX for zspr

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

ap COMPLEX for cspr

DOUBLE COMPLEX for zspr

Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry, with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *A*(1,1), *ap*(2) and *ap*(3) contain *A*(1,2) and *A*(2,2) respectively, and so on.

Before entry, with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(2,1) and *a*(3,1) respectively, and so on.

Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

?syconv

Converts a symmetric matrix given by a triangular matrix factorization into two matrices and vice versa.

Syntax

```
call ssyconv( uplo, way, n, a, lda, ipiv, e, info )
call dsyconv( uplo, way, n, a, lda, ipiv, e, info )
call csyconv( uplo, way, n, a, lda, ipiv, e, info )
call zsyconv( uplo, way, n, a, lda, ipiv, e, info )
call syconv( a[,uplo][,way][,ipiv][,info][,e] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine converts matrix *A*, which results from a triangular matrix factorization, into matrices *L* and *D* and vice versa. The routine returns non-diagonalized elements of *D* and applies or reverses permutation done with the triangular matrix factorization.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the details of the factorization are stored as an upper or lower triangular matrix:

If *uplo* = 'U': the upper triangular, $A = U*D*U^T$.

If *uplo* = 'L': the lower triangular, $A = L*D*L^T$.

way

CHARACTER*1. Must be 'C' or 'R'.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

a

REAL for ssyconv

DOUBLE PRECISION for dsyconv

COMPLEX for csyconv

DOUBLE COMPLEX for zsyconv

Array of size *lda* by *n*.

The block diagonal matrix *D* and the multipliers used to obtain the factor U or L as computed by ?sytrf.

lda

INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ipiv

INTEGER. Array, size at least max(1, *n*).

Details of the interchanges and the block structure of *D*, as returned by ?sytrf.

Output Parameters

e

REAL for ssyconv

DOUBLE PRECISION for dsyconv

COMPLEX for csyconv

DOUBLE COMPLEX for zsyconv

Array of size max(1, *n*) containing the superdiagonal/subdiagonal of the symmetric 1-by-1 or 2-by-2 block diagonal matrix *D* in $L*D*L^T$.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = –1011, memory allocation error occurred.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `syconv` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
uplo	Must be `'U'` or `'L'`.
way	Must be `'C'` or `'R'`.
ipiv	Holds the vector of length *n*.
e	Holds the vector of length *n*.

See Also

?sytrf

?symv

Computes a matrix-vector product for a complex symmetric matrix.

Syntax

```
call csymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call zsymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Include Files

- mkl.fi

Description

The routine performs the matrix-vector operation defined as

y := *alpha***a***x* + *beta***y*,

where:

alpha and *beta* are complex scalars

x and *y* are *n*-element complex vectors

a is an *n*-by-*n* symmetric complex matrix.

These routines have their real equivalents in BLAS (see ?symv in Chapter "BLAS and Sparse BLAS Routines").

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is used: If *uplo* = `'U'` or `'u'`, then the upper triangular part of the array *a* is used. If *uplo* = `'L'` or `'l'`, then the lower triangular part of the array *a* is used.
n	INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha, *beta*	COMPLEX for csymv DOUBLE COMPLEX for zsymv Specify the scalars *alpha* and *beta*. When *beta* is supplied as zero, then *y* need not be set on input.
a	COMPLEX for csymv DOUBLE COMPLEX for zsymv Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.
lda	INTEGER. Specifies the leading dimension of *A* as declared in the calling (sub)program. The value of *lda* must be at least max(1,*n*).
x	COMPLEX for csymv DOUBLE COMPLEX for zsymv Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	COMPLEX for csymv DOUBLE COMPLEX for zsymv Array, DIMENSION at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

?syr

Performs the symmetric rank-1 update of a complex symmetric matrix.

Syntax

```
call csyr( uplo, n, alpha, x, incx, a, lda )
call zsyr( uplo, n, alpha, x, incx, a, lda )
```

Include Files

- mkl.fi

Description

The routine performs the symmetric rank 1 operation defined as

$a := alpha*x*x^H + a$,

where:

- *alpha* is a complex scalar.
- *x* is an *n*-element complex vector.
- *a* is an *n*-by-*n* complex symmetric matrix.

These routines have their real equivalents in BLAS (see `?syr` in Chapter "BLAS and Sparse BLAS Routines").

Input Parameters

uplo

`CHARACTER*1`. Specifies whether the upper or lower triangular part of the array *a* is used:

If *uplo* = `'U'` or `'u'`, then the upper triangular part of the array *a* is used.

If *uplo* = `'L'` or `'l'`, then the lower triangular part of the array *a* is used.

n

`INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha

`COMPLEX` for `csyr`

`DOUBLE COMPLEX` for `zsyr`

Specifies the scalar *alpha*.

x

`COMPLEX` for `csyr`

`DOUBLE COMPLEX` for `zsyr`

Array, size at least `(1 + (n - 1)*abs(incx))`. Before entry, the incremented array *x* must contain the *n*-element vector `x`.

incx

`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

a

`COMPLEX` for `csyr`

`DOUBLE COMPLEX` for `zsyr`

Array, size `(lda, n)`. Before entry with *uplo* = `'U'` or `'u'`, the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = `'L'` or `'l'`, the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda

`INTEGER`. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1,n)`.

Output Parameters

a

With *uplo* = `'U'` or `'u'`, the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

i?max1

Finds the index of the vector element whose real part has maximum absolute value.

Syntax

index = icmax1(*n*, *cx*, *incx*)

index = izmax1(*n*, *cx*, *incx*)

Include Files

- mkl.fi

Description

Given a complex vector *cx*, the i?max1 functions return the index of the first vector element of maximum absolute value. These functions are based on the BLAS functions icamax/izamax, but using the absolute value of components. They are designed for use with clacon/zlacon.

Input Parameters

n — INTEGER. Specifies the number of elements in the vector *cx*.

cx — COMPLEX for icmax1

DOUBLE COMPLEX for izmax1

Array, size at least (1+(*n*-1)*abs(*incx*)).

Contains the input vector.

incx — INTEGER. Specifies the spacing between successive elements of *cx*.

Output Parameters

index — INTEGER. Index of the vector element of maximum absolute value.

?sum1

Forms the 1-norm of the complex vector using the true absolute value.

Syntax

res = scsum1(*n*, *cx*, *incx*)

res = dzsum1(*n*, *cx*, *incx*)

Include Files

- mkl.fi

Description

Given a complex vector *cx*, scsum1/dzsum1 functions take the sum of the absolute values of vector elements and return a single/double precision result, respectively. These functions are based on scasum/dzasum from Level 1 BLAS, but use the true absolute value and were designed for use with clacon/zlacon.

Input Parameters

n	INTEGER. Specifies the number of elements in the vector *cx*.
cx	COMPLEX for scsum1 DOUBLE COMPLEX for dzsum1 Array, size at least (1+(*n*-1)*abs(*incx*)). Contains the input vector whose elements will be summed.
incx	INTEGER. Specifies the spacing between successive elements of *cx* (*incx* > 0).

Output Parameters

res	REAL for scsum1 DOUBLE PRECISION for dzsum1 Sum of absolute values.

?gbtf2

Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.

Syntax

```
call sgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine forms the *LU* factorization of a general real/complex *m*-by-*n* band matrix *A* with *kl* sub-diagonals and *ku* super-diagonals. The routine uses partial pivoting with row interchanges and implements the unblocked version of the algorithm, calling Level 2 BLAS. See also ?gbtrf.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in *A* (*n*≥ 0).
kl	INTEGER. The number of sub-diagonals within the band of *A* (*kl*≥ 0).
ku	INTEGER. The number of super-diagonals within the band of *A* (*ku*≥ 0).
ab	REAL for sgbtf2 DOUBLE PRECISION for dgbtf2 COMPLEX for cgbtf2 DOUBLE COMPLEX for zgbtf2. Array, DIMENSION (*ldab*,*). The array *ab* contains the matrix *A* in band storage (see Matrix Arguments). The second dimension of *ab* must be at least max(1, *n*).
ldab	INTEGER. The leading dimension of the array *ab*. (*ldab*≥ 2*kl* + *ku* +1)

Output Parameters

ab	Overwritten by details of the factorization. The diagonal and *kl* + *ku* super-diagonals of *U* are stored in the first 1 + *kl* + *ku* rows of *ab*. The multipliers used during the factorization are stored in the next *kl* rows.
ipiv	INTEGER. Array, DIMENSION at least max(1,min(*m*,*n*)). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).
info	INTEGER. If *info* =0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

?gebd2

Reduces a general matrix to bidiagonal form using an unblocked algorithm.

Syntax

```
call sgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call dgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call cgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call zgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
```

Include Files

- mkl.fi

Description

The routine reduces a general *m*-by-*n* matrix *A* to upper or lower bidiagonal form *B* by an orthogonal (unitary) transformation: $Q^T*A*P = B$ (for real flavors) or $Q^H*A*P = B$ (for complex flavors).

If $m \geq n$, *B* is upper bidiagonal; if $m < n$, *B* is lower bidiagonal.

The routine does not form the matrices *Q* and *P* explicitly, but represents them as products of elementary reflectors. if $m \geq n$,

$Q = H(1)*H(2)*...*H(n)$, and $P = G(1)*G(2)*...*G(n-1)$

if $m < n$,

$Q = H(1)*H(2)*...*H(m-1)$, and $P = G(1)*G(2)*...*G(m)$

Each $H(i)$ and $G(i)$ has the form

$H(i) = I - tauq*v*v^T$ and $G(i) = I - taup*u*u^T$ for real flavors, or

$H(i) = I - tauq*v*v^H$ and $G(i) = I - taup*u*u^H$ for complex flavors

where *tauq* and *taup* are scalars (real for sgebd2/dgebd2, complex for cgebd2/zgebd2), and *v* and *u* are vectors (real for sgebd2/dgebd2, complex for cgebd2/zgebd2).

Input Parameters

m	INTEGER. The number of rows in the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq 0$).
a, *work*	REAL for sgebd2 DOUBLE PRECISION for dgebd2 COMPLEX for cgebd2 DOUBLE COMPLEX for zgebd2. Arrays: *a*(*lda*,*) contains the *m*-by-*n* general matrix *A* to be reduced. The second dimension of *a* must be at least max(1, *n*). *work*(*) is a workspace array, the dimension of *work* must be at least max(1, *m*, *n*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).

Output Parameters

a	if $m \geq n$, the diagonal and first super-diagonal of *a* are overwritten with the upper bidiagonal matrix *B*. Elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements above the first superdiagonal, with the array *taup*, represent the orthogonal/unitary matrix *p* as a product of elementary reflectors.

if *m* < *n*, the diagonal and first sub-diagonal of *a* are overwritten by the lower bidiagonal matrix *B*. Elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements above the diagonal, with the array *taup*, represent the orthogonal/unitary matrix *p* as a product of elementary reflectors.

d

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION at least max(1, min(*m*, *n*)).

Contains the diagonal elements of the bidiagonal matrix *B*: *d*(i) = *a*(i, i).

e

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, min(*m*, *n*) - 1).

Contains the off-diagonal elements of the bidiagonal matrix *B*:

if *m*≥*n*, *e*(i) = *a*(i, i+1) for i = 1,2,..., *n*-1;

if *m* < *n*, *e*(i) = *a*(i+1, i) for i = 1,2,..., *m*-1.

tauq, *taup*

REAL for sgebd2

DOUBLE PRECISION for dgebd2

COMPLEX for cgebd2

DOUBLE COMPLEX for zgebd2.

Arrays, DIMENSION at least max (1, min(*m*, *n*)).

Contain scalar factors of the elementary reflectors which represent orthogonal/unitary matrices *Q* and *p*, respectively.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

?gehd2

Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.

Syntax

```
call sgehd2( n, ilo, ihi, a, lda, tau, work, info )
call dgehd2( n, ilo, ihi, a, lda, tau, work, info )
call cgehd2( n, ilo, ihi, a, lda, tau, work, info )
call zgehd2( n, ilo, ihi, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine reduces a real/complex general matrix *A* to upper Hessenberg form *H* by an orthogonal or unitary similarity transformation $Q^T*A*Q = H$ (for real flavors) or $Q^H*A*Q = H$ (for complex flavors).

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of *elementary reflectors*.

Input Parameters

n	INTEGER The order of the matrix *A* ($n \geq 0$).
ilo, *ihi*	INTEGER. It is assumed that *A* is already upper triangular in rows and columns 1:*ilo* −1 and *ihi*+1:*n*. If *A* has been output by ?gebal, then *ilo* and *ihi* must contain the values returned by that routine. Otherwise they should be set to $ilo = 1$ and $ihi = n$. Constraint: $1 \leq ilo \leq ihi \leq \max(1, n)$.
a, *work*	REAL for sgehd2 DOUBLE PRECISION for dgehd2 COMPLEX for cgehd2 DOUBLE COMPLEX for zgehd2. Arrays: *a* (*lda*,*) contains the *n*-by-*n* matrix *A* to be reduced. The second dimension of *a* must be at least max(1, *n*). *work* (*n*) is a workspace array.
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).

Output Parameters

a	On exit, the upper triangle and the first subdiagonal of *A* are overwritten with the upper Hessenberg matrix *H* and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. See *Application Notes* below.
tau	REAL for sgehd2 DOUBLE PRECISION for dgehd2 COMPLEX for cgehd2 DOUBLE COMPLEX for zgehd2. Array, DIMENSION at least max (1, *n*−1). Contains the scalar factors of elementary reflectors. See *Application Notes* below.
info	INTEGER. If $info = 0$, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The matrix *Q* is represented as a product of (*ihi* - *ilo*) elementary reflectors

`Q = H(ilo)*H(ilo +1)*...*H(ihi -1)`

Each *H*(i) has the form

$H(i) = I - tau*v*v^T$ for real flavors, or

$H(i) = I - tau*v*v^H$ for complex flavors

where *tau* is a real/complex scalar, and *v* is a real/complex vector with `v(1:i) = 0`, `v(i+1) = 1` and `v(ihi +1:n) = 0`.

On exit, `v(i+2:ihi)` is stored in `a(i+2:ihi, i)` and *tau* in `tau(i)`.

The contents of *a* are illustrated by the following example, with `n = 7`, `ilo = 2` and `ihi = 6`:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & & & & & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & h & h & h & h & a \\ & a & h & h & h & h & a \\ & h & h & h & h & h & h \\ & v_2 & h & h & h & h & h \\ & v_2 & v_3 & h & h & h & h \\ & v_2 & v_3 & v_4 & h & h & h \\ & & & & & & a \end{bmatrix}$$

where *a* denotes an element of the original matrix *A*, *h* denotes a modified element of the upper Hessenberg matrix *H*, and v_i denotes an element of the vector defining *H*(i).

?gelq2

Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgelq2( m, n, a, lda, tau, work, info )
call dgelq2( m, n, a, lda, tau, work, info )
call cgelq2( m, n, a, lda, tau, work, info )
call zgelq2( m, n, a, lda, tau, work, info )
```

Include Files

- `mkl.fi`

Description

The routine computes an *LQ* factorization of a real/complex *m*-by-*n* matrix *A* as $A = L*Q$.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors* :

$Q = H(k) \ldots H(2) H(1)$ (or $Q = H(k)^H \ldots H(2)^H H(1)^H$ for complex flavors), where $k = \min(m, n)$

Each *H*(i) has the form

$H(i) = I - tau*v*v^T$ for real flavors, or

$H(i) = I - tau*v*v^H$ for complex flavors,

where *tau* is a real/complex scalar stored in *tau*(i), and *v* is a real/complex vector with $v_{1:i-1} = 0$ and $v_i = 1$.

On exit, $v_{i+1:n}$ (for real functions) and $\mathrm{conjg}(v_{i+1:n})$ (for complex functions) are stored in $a(i, i+1:n)$.

Input Parameters

The data types are given for the Fortran interface.

m
: INTEGER. The number of rows in the matrix *A* ($m \geq 0$).

n
: INTEGER. The number of columns in *A* ($n \geq 0$).

a, *work*
: REAL for sgelq2

  DOUBLE PRECISION for dgelq2

  COMPLEX for cgelq2

  DOUBLE COMPLEX for zgelq2.

  Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least $\max(1, n)$.

  work(*m*) is a workspace array.

lda
: INTEGER. The leading dimension of *a*; at least max(1, *m*) .

Output Parameters

a
: Overwritten by the factorization data as follows:

  on exit, the elements on and below the diagonal of the array *a* contain the *m*-by-min(*n*,*m*) lower trapezoidal matrix *L* (*L* is lower triangular if $n \geq m$); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors.

tau
: REAL for sgelq2

  DOUBLE PRECISION for dgelq2

  COMPLEX for cgelq2

  DOUBLE COMPLEX for zgelq2.

  Array, size at least $\max(1, \min(m, n))$.

  Contains scalar factors of the elementary reflectors.

info
: INTEGER.

  If $info = 0$, the execution is successful.

  If $info = -i$, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?gelqt3

?gelqt3 *recursively computes a LQ factorization of a general real or complex M-by-N matrix A, using the compact WY representation of Q.*

```
call sgelqt3(m, n, a, lda, t, ldt, info)
call dgelqt3(m, n, a, lda, t, ldt, info)
call cgelqt3(m, n, a, lda, t, ldt, info)
call zgelqt3(m, n, a, lda, t, ldt, info)
```

Description

?gelqt3 recursively computes a LQ factorization of a real or complex *m*-by-*n* matrix *A*, using the compact WY representation of *Q*. Based on the algorithm of Elmroth and Gustavson [ELMROTH00].

The matrix *V* stores the elementary reflectors *H*(*i*) in the *i*-th row above the diagonal. For example, if *m*=5 and *n*=3, the matrix *V* is

$$V = \begin{pmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{pmatrix}$$

where the v_is represent the vectors which define *H*(*i*), which are returned in the array *a*. The 1 elements along the diagonal of *V* are not stored in *a*. The block reflector *H* is then given by

$H = I - V * T * V^T$ for real matrices, or

$H = I - V * T * V^H$ for complex matrices.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. $m \le n$.
n	INTEGER. The number of columns of the matrix *A*. $n \ge 0$.
a	REAL for sgelqt3 DOUBLE PRECISION for dgelqt3 COMPLEX for cgelqt3 COMPLEX*16 for zgelqt3 Array of size (*lda*,*n*). On entry, the real or complex *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. $lda \ge \max(1,m)$.
ldt	INTEGER. The leading dimension of the array *t*. $ldt \ge \max(1,n)$.

Output Parameters

a	On exit, the elements on and below the diagonal contain the *n*-by-*n* lower triangular matrix *L*; the elements above the diagonal are the rows of *V*. See Description for further details.
t	REAL for sgelqt3

DOUBLE PRECISION for dgelqt3

COMPLEX for cgelqt3

COMPLEX*16 for zgelqt3

Array of size (*ldt*,*n*). The *n*-by-*n* upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector *T*; the elements below the diagonal are not used. See Description for further details.

info — INTEGER.

info = 0: successful exit.

info < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?geql2

Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeql2( m, n, a, lda, tau, work, info )
call dgeql2( m, n, a, lda, tau, work, info )
call cgeql2( m, n, a, lda, tau, work, info )
call zgeql2( m, n, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine computes a *QL* factorization of a real/complex *m*-by-*n* matrix *A* as A = Q*L.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors* :

Q = H(k)* ... *H(2)*H(1), where k = min(m, n).

Each *H*(i) has the form

H(i) = I - tau*v*v^T for real flavors, or

H(i) = I - tau*v*v^H for complex flavors

where *tau* is a real/complex scalar stored in *tau*(i), and *v* is a real/complex vector with v(m-k+i+1:m) = 0 and v(m-k+i) = 1.

On exit, v(1:m-k+i-1) is stored in a(1:m-k+i-1, n-k+i).

Input Parameters

m — INTEGER. The number of rows in the matrix *A* (m≥ 0).

n — INTEGER. The number of columns in *A* (n≥ 0).

a, *work* — REAL for sgeql2

DOUBLE PRECISION for dgeql2

COMPLEX for cgeql2

DOUBLE COMPLEX for zgeql2.

Arrays:

a(*lda*,*) contains the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

work(*m*) is a workspace array.

lda INTEGER. The leading dimension of *a*; at least max(1, *m*).

Output Parameters

a Overwritten by the factorization data as follows:

on exit, if $m \geq n$, the lower triangle of the subarray *a*(*m*-*n*+1:*m*, 1:*n*) contains the *n*-by-*n* lower triangular matrix *L*; if $m < n$, the elements on and below the (*n*-*m*)th superdiagonal contain the *m*-by-*n* lower trapezoidal matrix *L*; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

tau REAL for sgeql2

DOUBLE PRECISION for dgeql2

COMPLEX for cgeql2

DOUBLE COMPLEX for zgeql2.

Array, DIMENSION at least max(1, min(*m*, *n*)).

Contains scalar factors of the elementary reflectors.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?geqr2

Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeqr2( m, n, a, lda, tau, work, info )
call dgeqr2( m, n, a, lda, tau, work, info )
call cgeqr2( m, n, a, lda, tau, work, info )
call zgeqr2( m, n, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine computes a *QR* factorization of a real/complex *m*-by-*n* matrix *A* as $A = Q*R$.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors* :

$Q = H(1)*H(2)* \ldots *H(k)$, where $k = \min(m, n)$

Each *H*(i) has the form

$H(i) = I - tau*v*v^T$ for real flavors, or

$H(i) = I - tau*v*v^H$ for complex flavors

where *tau* is a real/complex scalar stored in *tau*`(i)`, and *v* is a real/complex vector with $v_{1:i-1} = 0$ and $v_i = 1$.

On exit, $v_{i+1:m}$ is stored in *a*`(i+1:`*m*`, i)`.

Input Parameters

The data types are given for the Fortran interface.

m	INTEGER. The number of rows in the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq 0$).
a, *work*	REAL for sgeqr2 DOUBLE PRECISION for dgeqr2 COMPLEX for cgeqr2 DOUBLE COMPLEX for zgeqr2. Arrays: *a*`(lda,*)` contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least `max(1, n)`. *work*(*n*) is a workspace array.
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*) .

Output Parameters

a	Overwritten by the factorization data as follows: on exit, the elements on and above the diagonal of the array *a* contain the min(*n*,*m*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if $m \geq n$); the elements below the diagonal, with the array *tau*, represent the orthogonal/ unitary matrix *Q* as a product of elementary reflectors.
tau	REAL for sgeqr2 DOUBLE PRECISION for dgeqr2 COMPLEX for cgeqr2 DOUBLE COMPLEX for zgeqr2. Array, size at least `max(1, min(`*m*`, n))`. Contains scalar factors of the elementary reflectors.
info	INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?geqr2p

Computes the QR factorization of a general rectangular matrix with non-negative diagonal elements using an unblocked algorithm.

Syntax

```
call sgeqr2p( m, n, a, lda, tau, work, info )
call dgeqr2p( m, n, a, lda, tau, work, info )
call cgeqr2p( m, n, a, lda, tau, work, info )
call zgeqr2p( m, n, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine computes a *QR* factorization of a real/complex *m*-by-*n* matrix *A* as *A* = *Q***R*. The diagonal entries of *R* are real and nonnegative.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors* :

Q = *H*(1)**H*(2)* ... **H*(k), where *k* = min(*m*, *n*)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*^{T} for real flavors, or

H(i) = *I* - *tau***v***v*^{H} for complex flavors

where *tau* is a real/complex scalar stored in *tau*(i), and *v* is a real/complex vector with *v*(1:i-1) = 0 and *v*(i) = 1.

On exit, *v*(i+1:*m*) is stored in *a*(i+1:*m*, i).

Input Parameters

m — INTEGER. The number of rows in the matrix *A* (*m*≥ 0).

n — INTEGER. The number of columns in *A* (*n*≥ 0).

a, *work* — REAL for sgeqr2p

DOUBLE PRECISION for d

COMPLEX for cgeqr2p

DOUBLE COMPLEX for zgeqr2p.

Arrays:

a(*lda*,*) contains the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least `max(1, n)`.

work(*n*) is a workspace array.

lda — `INTEGER`. The leading dimension of *a*; at least `max(1, m)`.

Output Parameters

a — Overwritten by the factorization data as follows:

on exit, the elements on and above the diagonal of the array *a* contain the min(*n*,*m*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if $m \geq n$). The diagonal entries of *R* are real and nonnegative; the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

tau — `REAL` for `sgeqr2p`

`DOUBLE PRECISION` for `dgeqr2p`

`COMPLEX` for `cgeqr2p`

`DOUBLE COMPLEX` for `zgeqr2p`.

Array, `DIMENSION` at least `max(1, min(m, n))`.

Contains scalar factors of the elementary reflectors.

info — `INTEGER`.

If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.

?geqrt2

Computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.

Syntax

```
call sgeqrt2(m, n, a, lda, t, ldt, info)
call dgeqrt2(m, n, a, lda, t, ldt, info)
call cgeqrt2(m, n, a, lda, t, ldt, info)
call zgeqrt2(m, n, a, lda, t, ldt, info)
call geqrt2(a, t, [info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The strictly lower triangular matrix *V* contains the elementary reflectors *H*(*i*) in the *i*th column below the diagonal. For example, if m=5 and n=3, the matrix *V* is

$$V = \begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

where v_i represents the vector that defines $H(i)$. The vectors are returned in the lower triangular part of array `a`.

> **NOTE**
> The 1s along the diagonal of V are not stored in `a`.

The block reflector H is then given by

$H = I - V*T*V^T$ for real flavors, and

$H = I - V*T*V^H$ for complex flavors,

where V^T is the transpose and V^H is the conjugate transpose of V.

Input Parameters

`m`	`INTEGER`. The number of rows in the matrix A ($m \geq n$).
`n`	`INTEGER`. The number of columns in A ($n \geq 0$).

`a`	`REAL` for `sgeqrt2`
	`DOUBLE PRECISION` for `dgeqrt2`
	`COMPLEX` for `cgeqrt2`
	`COMPLEX*16` for `zgeqrt2`.
	Array, size `lda` by `n`. Array `a` contains the m-by-n matrix A.

`lda`	`INTEGER`. The leading dimension of `a`; at least max(1, m).
`ldt`	`INTEGER`. The leading dimension of `t`; at least max(1, `n`).

Output Parameters

`a`	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the `n`-by-`n` upper triangular matrix R. The elements below the diagonal are the columns of V.

`t`	`REAL` for `sgeqrt2`
	`DOUBLE PRECISION` for `dgeqrt2`
	`COMPLEX` for `cgeqrt2`
	`COMPLEX*16` for `zgeqrt2`.
	Array, size (`ldt`, min(`m`, `n`)).
	The `n`-by-`n` upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector T. The elements below the diagonal are not used.

`info`	`INTEGER`. If `info` = 0, the execution is successful. If `info` < 0 and `info` = -i, the ith argument had an illegal value. `If info = -1011`, memory allocation error occurred.

?geqrt3

Recursively computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.

Syntax

```
call sgeqrt3(m, n, a, lda, t, ldt, info)
call dgeqrt3(m, n, a, lda, t, ldt, info)
call cgeqrt3(m, n, a, lda, t, ldt, info)
call zgeqrt3(m, n, a, lda, t, ldt, info)
call geqrt3(a, t [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The strictly lower triangular matrix *V* contains the elementary reflectors *H*(*i*) in the *i*th column below the diagonal. For example, if *m*=5 and *n*=3, the matrix *V* is

$$V = \begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

where v_i represents one of the vectors that define $H(i)$. The vectors are returned in the lower part of triangular array `a`.

NOTE
The 1s along the diagonal of V are not stored in `a`.

The block reflector H is then given by

$H = I - V*T*V^T$ for real flavors, and

$H = I - V*T*V^H$ for complex flavors,

where V^T is the transpose and $V^H$is the conjugate transpose of V.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ *n*).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a	REAL for sgeqrt3 DOUBLE PRECISION for dgeqrt3 COMPLEX for cgeqrt3 COMPLEX*16 for zgeqrt3. Array, size (*lda*, *n*). Array *a* contains the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).
ldt	INTEGER. The leading dimension of *t*; at least max(1, *n*).

Output Parameters

a	The elements on and above the diagonal of the array contain the *n*-by-*n* upper triangular matrix *R*. The elements below the diagonal are the columns of *V*.
t	REAL for sgeqrt3 DOUBLE PRECISION for dgeqrt3 COMPLEX for cgeqrt3 COMPLEX*16 for zgeqrt3. Array, size *ldt* by *n*. The *n*-by-*n* upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector *T*. The elements below the diagonal are not used.
info	INTEGER. If *info* = 0, the execution is successful. If *info* < 0 and *info* = -*i*, the *i*th argument had an illegal value. If *info* = -1011, memory allocation error occurred.

?gerq2

Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgerq2( m, n, a, lda, tau, work, info )
call dgerq2( m, n, a, lda, tau, work, info )
call cgerq2( m, n, a, lda, tau, work, info )
call zgerq2( m, n, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine computes a *RQ* factorization of a real/complex *m*-by-*n* matrix *A* as $A = R*Q$.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors* :

$Q = H(1)*H(2)* \ldots *H(k)$ for real flavors, or

$Q = H(1)^H*H(2)^H* \ldots *H(k)^H$ for complex flavors

where $k = \min(m, n)$.

Each *H*(i) has the form

$H(i) = I - tau*v*v^T$ for real flavors, or

$H(i) = I - tau*v*v^H$ for complex flavors

where *tau* is a real/complex scalar stored in *tau*(i), and *v* is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$.

On exit, $v(1:n-k+i-1)$ is stored in $a(m-k+i, 1:n-k+i-1)$.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq 0$).
a, *work*	REAL for sgerq2 DOUBLE PRECISION for dgerq2 COMPLEX for cgerq2 DOUBLE COMPLEX for zgerq2. Arrays: $a(lda,*)$ contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least $\max(1, n)$. *work*(*m*) is a workspace array.
lda	INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

Output Parameters

a	Overwritten by the factorization data as follows: on exit, if $m \leq n$, the upper triangle of the subarray $a(1:m, n-m+1:n)$ contains the *m*-by-*m* upper triangular matrix *R*; if $m > n$, the elements on and above the (*m*-*n*)-th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.
tau	REAL for sgerq2

DOUBLE PRECISION for dgerq2

COMPLEX for cgerq2

DOUBLE COMPLEX for zgerq2.

Array, DIMENSION at least max(1, min(*m*, *n*)).

Contains scalar factors of the elementary reflectors.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?gesc2

Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2.

Syntax

```
call sgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call dgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call cgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call zgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
```

Include Files

- mkl.fi

Description

The routine solves a system of linear equations

*A*X = scale*RHS*

with a general *n*-by-*n* matrix *A* using the *LU* factorization with complete pivoting computed by ?getc2.

Input Parameters

n INTEGER. The order of the matrix *A*.

a, *rhs* REAL for sgesc2

DOUBLE PRECISION for dgesc2

COMPLEX for cgesc2

DOUBLE COMPLEX for zgesc2.

Arrays:

a(*lda*,*) contains the *LU* part of the factorization of the *n*-by-*n* matrix *A* computed by ?getc2:

*A = P*L*U*Q*.

The second dimension of *a* must be at least max(1, *n*);

rhs(*n*) contains on entry the right hand side vector for the system of equations.

lda INTEGER. The leading dimension of *a*; at least max(1, *n*).

ipiv INTEGER.

Array, DIMENSION at least max(1,*n*).

The pivot indices: for 1 ≤ i ≤ *n*, row i of the matrix has been interchanged with row *ipiv*(i).

jpiv INTEGER.

Array, DIMENSION at least max(1,*n*).

The pivot indices: for 1 ≤ j ≤ *n*, column j of the matrix has been interchanged with column *jpiv*(j).

Output Parameters

rhs On exit, overwritten with the solution vector *X*.

scale REAL for sgesc2/cgesc2

DOUBLE PRECISION for dgesc2/zgesc2

Contains the scale factor. *scale* is chosen in the range 0 ≤ *scale* ≤ 1 to prevent overflow in the solution.

?getc2

Computes the LU factorization with complete pivoting of the general n-by-n matrix.

Syntax

```
call sgetc2( n, a, lda, ipiv, jpiv, info )
call dgetc2( n, a, lda, ipiv, jpiv, info )
call cgetc2( n, a, lda, ipiv, jpiv, info )
call zgetc2( n, a, lda, ipiv, jpiv, info )
```

Include Files

- mkl.fi

Description

The routine computes an *LU* factorization with complete pivoting of the *n*-by-*n* matrix *A*. The factorization has the form *A = P*L*U*Q*, where *P* and *Q* are permutation matrices, *L* is lower triangular with unit diagonal elements and *U* is upper triangular.

The LU factorization computed by this routine is used by ?latdf to compute a contribution to the reciprocal Dif-estimate.

Input Parameters

n	INTEGER. The order of the matrix *A* ($n \geq 0$).
a	REAL for sgetc2 DOUBLE PRECISION for dgetc2 COMPLEX for cgetc2 DOUBLE COMPLEX for zgetc2. Array *a*(*lda*,*) contains the *n*-by-*n* matrix *A* to be factored. The second dimension of *a* must be at least max(1, *n*);
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).

Output Parameters

a	On exit, the factors *L* and *U* from the factorization *A = P*L*U*Q*; the unit diagonal elements of *L* are not stored. If *U*(*k*, *k*) appears to be less than smin, *U*(*k*, *k*) is given the value of smin, that is giving a nonsingular perturbed system.
ipiv	INTEGER. Array, DIMENSION at least max(1,*n*). The pivot indices: for $1 \leq i \leq n$, row i of the matrix has been interchanged with row *ipiv*(i).
jpiv	INTEGER. Array, DIMENSION at least max(1,*n*). The pivot indices: for $1 \leq j \leq n$, column j of the matrix has been interchanged with column *jpiv*(j).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *k*>0, *U*(*k*, *k*) is likely to produce overflow if we try to solve for *x* in *A*x = b*. So *U* is perturbed to avoid the overflow.

?getf2

Computes the LU factorization of a general m-by-n matrix using partial pivoting with row interchanges (unblocked algorithm).

Syntax

```
call sgetf2( m, n, a, lda, ipiv, info )
call dgetf2( m, n, a, lda, ipiv, info )
call cgetf2( m, n, a, lda, ipiv, info )
call zgetf2( m, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine computes the *LU* factorization of a general *m*-by-*n* matrix *A* using partial pivoting with row interchanges. The factorization has the form

*A = P*L*U*

where *p* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if *m* > *n*) and *U* is upper triangular (upper trapezoidal if *m* < *n*).

Input Parameters

The data types are given for the Fortran interface.

m	INTEGER. The number of rows in the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in *A* (*n*≥ 0).
a	REAL for sgetf2 DOUBLE PRECISION for dgetf2 COMPLEX for cgetf2 DOUBLE COMPLEX for zgetf2. Array, size (*lda*,*). Contains the matrix *A* to be factored. The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The leading dimension of *a*; at least max(1, *m*).

Output Parameters

a	Overwritten by *L* and *U*. The unit diagonal elements of *L* are not stored.
ipiv	INTEGER. Array, size at least max(1,min(*m*,*n*)). The pivot indices: for 1 ≤ i ≤ *n*, row *i* was interchanged with row *ipiv*(*i*).
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i* >0, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations. If *info* = -1011, memory allocation error occurred.

?gtts2

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.

Syntax

```
call sgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call dgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call cgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call zgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
```

Include Files

- mkl.fi

Description

The routine solves for *X* one of the following systems of linear equations with multiple right hand sides:

*A*X = B*, $A^T*X = B$, or $A^H*X = B$ (for complex matrices only), with a tridiagonal matrix A using the *LU* factorization computed by ?gttrf.

Input Parameters

itrans

INTEGER. Must be 0, 1, or 2.

Indicates the form of the equations to be solved:

If *itrans* = 0, then *A*X = B* (no transpose).

If *itrans* = 1, then $A^T*X = B$ (transpose).

If *itrans* = 2, then $A^H*X = B$ (conjugate transpose).

n

INTEGER. The order of the matrix *A* (*n*≥ 0).

nrhs

INTEGER. The number of right-hand sides, i.e., the number of columns in *B* (*nrhs*≥ 0).

dl,*d*,*du*,*du2*,*b*

REAL for sgtts2

DOUBLE PRECISION for dgtts2

COMPLEX for cgtts2

DOUBLE COMPLEX for zgtts2.

Arrays: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1), *du2*(*n* - 2), *b*(*ldb*,*nrhs*).

The array *dl* contains the (*n* - 1) multipliers that define the matrix *L* from the *LU* factorization of *A*.

The array *d* contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

The array *du* contains the (*n* - 1) elements of the first super-diagonal of *U*.

The array *du2* contains the (*n* - 2) elements of the second super-diagonal of *U*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldb

INTEGER. The leading dimension of *b*; must be *ldb*≥ max(1, *n*).

ipiv

INTEGER.

Array, DIMENSION (*n*).

The pivot indices array, as returned by ?gttrf.

Output Parameters

b Overwritten by the solution matrix *X*.

?isnan

Tests input for NaN.

Syntax

val = sisnan(*sin*)

val = disnan(*din*)

Include Files

- mkl.fi

Description

This logical routine returns .TRUE. if its argument is NaN, and .FALSE. otherwise.

Input Parameters

sin REAL for sisnan

Input to test for NaN.

din DOUBLE PRECISION for disnan

Input to test for NaN.

Output Parameters

val Logical. Result of the test.

?laisnan

Tests input for NaN.

Syntax

val = slaisnan(*sin1*, *sin2*)

val = dlaisnan(*din1*, *din2*)

Include Files

- mkl.fi

Description

This logical routine checks for NaNs (NaN stands for 'Not A Number') by comparing its two arguments for inequality. NaN is the only floating-point value where NaN ≠ NaN returns .TRUE. To check for NaNs, pass the same variable as both arguments.

This routine is not for general use. It exists solely to avoid over-optimization in ?isnan.

Input Parameters

sin1, *sin2*	REAL for sisnan Two numbers to compare for inequality.
din2, *din2*	DOUBLE PRECISION for disnan Two numbers to compare for inequality.

Output Parameters

val	Logical. Result of the comparison.

?labrd

Reduces the first nb rows and columns of a general matrix to a bidiagonal form.

Syntax

```
call slabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call dlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call clabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call zlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
```

Include Files

- mkl.fi

Description

The routine reduces the first *nb* rows and columns of a general *m*-by-*n* matrix *A* to upper or lower bidiagonal form by an orthogonal/unitary transformation *Q'*A*P*, and returns the matrices *X* and *Y* which are needed to apply the transformation to the unreduced part of *A*.

if $m \geq n$, *A* is reduced to upper bidiagonal form; if $m < n$, to lower bidiagonal form.

The matrices *Q* and *P* are represented as products of elementary reflectors: $Q = H(1)*(2)* \ldots *H(nb)$, and $P = G(1)*G(2)* \ldots *G(nb)$

Each *H*(i) and *G*(i) has the form

$H(i) = I - tauq*v*v'$ and $G(i) = I - taup*u*u'$

where *tauq* and *taup* are scalars, and *v* and *u* are vectors.

The elements of the vectors *v* and *u* together form the *m*-by-*nb* matrix *V* and the *nb*-by-*n* matrix *U'* which are needed, with *X* and *Y*, to apply the transformation to the unreduced part of the matrix, using a block update of the form: $A := A - V*Y' - X*U'$.

This is an auxiliary routine called by ?gebrd.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* ($m \geq 0$).

n INTEGER. The number of columns in *A* (*n*≥ 0).

nb INTEGER. The number of leading rows and columns of *A* to be reduced.

a REAL for slabrd

DOUBLE PRECISION for dlabrd

COMPLEX for clabrd

DOUBLE COMPLEX for zlabrd.

Array *a*(*lda*,*) contains the matrix *A* to be reduced. The second dimension of *a* must be at least max(1, *n*).

lda INTEGER. The leading dimension of *a*; at least max(1, *m*).

ldx INTEGER. The leading dimension of the output array *x*; must beat least max(1, *m*).

ldy INTEGER. The leading dimension of the output array *y*; must beat least max(1, *n*).

Output Parameters

a On exit, the first *nb* rows and columns of the matrix are overwritten; the rest of the array is unchanged.

if *m*≥*n*, elements on and below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors; and elements above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *p* as a product of elementary reflectors.

if *m* < *n*, elements below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements on and above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *p* as a product of elementary reflectors.

d, *e* REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors. Arrays, DIMENSION (*nb*) each. The array *d* contains the diagonal elements of the first *nb* rows and columns of the reduced matrix:

d(i) = *a*(i,i).

The array *e* contains the off-diagonal elements of the first *nb* rows and columns of the reduced matrix.

tauq, *taup* REAL for slabrd

DOUBLE PRECISION for dlabrd

COMPLEX for clabrd

DOUBLE COMPLEX for zlabrd.

Arrays, DIMENSION (*nb*) each. Contain scalar factors of the elementary reflectors which represent the orthogonal/unitary matrices *Q* and *P*, respectively.

x, *y*	`REAL` for `slabrd`
	`DOUBLE PRECISION` for `dlabrd`
	`COMPLEX` for `clabrd`
	`DOUBLE COMPLEX` for `zlabrd`.
	Arrays, dimension *x*(*ldx*, *nb*), *y*(*ldy*, *nb*).
	The array *x* contains the *m*-by-*nb* matrix *X* required to update the unreduced part of *A*.
	The array *y* contains the *n*-by-*nb* matrix *Y* required to update the unreduced part of *A*.

Application Notes

if $m \geq n$, then for the elementary reflectors *H*(i) and *G*(i),

`v(1:i-1) = 0`, `v(i) = 1`, and `v(i:m)` is stored on exit in `a(i:m, i)`; `u(1:i) = 0`, `u(i+1) = 1`, and `u(i+1:n)` is stored on exit in `a(i, i+1:n)`;

tauq is stored in *tauq*(i) and *taup* in *taup*(i).

if $m < n$,

`v(1:i) = 0`, `v(i+1) = 1`, and `v(i+1:m)` is stored on exit in `a(i+2:m, i)`; `u(1:i-1) = 0`, `u(i) = 1`, and `u(i:n)` is stored on exit in `a(i, i+1:n)`; *tauq* is stored in *tauq*(i) and *taup* in *taup*(i).

The contents of *a* on exit are illustrated by the following examples with `nb = 2`:

$m = 6,\ n = 5\ (m > n)$

$$\begin{bmatrix} 1 & 1 & u_1 & u_1 & u_1 \\ v_1 & 1 & 1 & u_2 & u_2 \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

$m = 5,\ n = 6\ (m < n)$

$$\begin{bmatrix} 1 & u_1 & u_1 & u_1 & u_1 & u_1 \\ 1 & 1 & u_2 & u_2 & u_2 & u_2 \\ v_1 & 1 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \end{bmatrix}$$

where *a* denotes an element of the original matrix which is unchanged, v_i denotes an element of the vector defining *H*(i), and u_i an element of the vector defining *G*(i).

?lacn2

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacn2( n, v, x, isgn, est, kase, isave )
call dlacn2( n, v, x, isgn, est, kase, isave )
call clacn2( n, v, x, est, kase, isave )
```

```
call zlacn2( n, v, x, est, kase, isave )
```

Include Files

- mkl.fi

Description

The routine estimates the 1-norm of a square, real or complex matrix *A*. Reverse communication is used for evaluating matrix-vector products.

Input Parameters

n	INTEGER. The order of the matrix *A* ($n \geq 1$).
v, *x*	REAL for slacn2 DOUBLE PRECISION for dlacn2 COMPLEX for clacn2 DOUBLE COMPLEX for zlacn2. Arrays, size (*n*) each. *v* is a workspace array. *x* is used as input after an intermediate return.
isgn	INTEGER. Workspace array, size (*n*), used with real flavors only.
est	REAL for slacn2/clacn2 DOUBLE PRECISION for dlacn2/zlacn2 On entry with *kase* set to 1 or 2, and *isave*(1) = 1, *est* must be unchanged from the previous call to the routine.
kase	INTEGER. On the initial call to the routine, *kase* must be set to 0.
isave	INTEGER. Array, size (*3*). Contains variables from the previous call to the routine.

Output Parameters

est	An estimate (a lower bound) for norm(*A*).
kase	On an intermediate return, *kase* is set to 1 or 2, indicating whether *x* is overwritten by $A*x$ or A^T*x for real flavors and $A*x$ or A^H*x for complex flavors. On the final return, *kase* is set to 0.
v	On the final return, *v* = *A***w*, where *est* = norm(*v*)/norm(*w*) (*w* is not returned).
x	On an intermediate return, *x* is overwritten by *A***x*, if *kase* = 1,

A^T*x, if *kase* = 2 (for real flavors),

A^H*x, if *kase* = 2 (for complex flavors),

and the routine must be re-called with all the other parameters unchanged.

isave This parameter is used to save variables between calls to the routine.

?lacon

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacon( n, v, x, isgn, est, kase )
call dlacon( n, v, x, isgn, est, kase )
call clacon( n, v, x, est, kase )
call zlacon( n, v, x, est, kase )
```

Include Files

- mkl.fi

Description

The routine estimates the 1-norm of a square, real/complex matrix *A*. Reverse communication is used for evaluating matrix-vector products.

WARNING
The ?lacon routine is not thread-safe. It is deprecated and retained for the backward compatibility only. Use the thread-safe ?lacn2 routine instead.

Input Parameters

n INTEGER. The order of the matrix *A* ($n \geq 1$).

v, *x* REAL for slacon

DOUBLE PRECISION for dlacon

COMPLEX for clacon

DOUBLE COMPLEX for zlacon.

Arrays, DIMENSION (*n*) each.

v is a workspace array.

x is used as input after an intermediate return.

isgn INTEGER.

Workspace array, DIMENSION (*n*), used with real flavors only.

est REAL for slacon/clacon

DOUBLE PRECISION for dlacon/zlacon

An estimate that with *kase*=1 or 2 should be unchanged from the previous call to ?lacon.

kase INTEGER.

On the initial call to ?lacon, *kase* should be 0.

Output Parameters

est REAL for slacon/clacon

DOUBLE PRECISION for dlacon/zlacon

An estimate (a lower bound) for norm(*A*).

kase On an intermediate return, *kase* will be 1 or 2, indicating whether *x* should be overwritten by *A***x* or A^T**x* for real flavors and *A***x* or A^H**x* for complex flavors. On the final return from ?lacon, *kase* will again be 0.

v On the final return, *v* = *A***w*, where *est* = norm(*v*)/norm(*w*) (*w* is not returned).

x On an intermediate return, *x* should be overwritten by

*A***x*, if *kase* = 1,

A^T**x*, if *kase* = 2 (for real flavors),

A^H**x*, if *kase* = 2 (for complex flavors),

and ?lacon must be re-called with all the other parameters unchanged.

?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call slacpy( uplo, m, n, a, lda, b, ldb )
call dlacpy( uplo, m, n, a, lda, b, ldb )
call clacpy( uplo, m, n, a, lda, b, ldb )
call zlacpy( uplo, m, n, a, lda, b, ldb )
```

Include Files

- mkl.fi

Description

The routine copies all or part of a two-dimensional matrix *A* to another matrix *B*.

Input Parameters

The data types are given for the Fortran interface.

uplo CHARACTER*1.

Specifies the part of the matrix *A* to be copied to *B*.

If *uplo* = 'U', the upper triangular part of *A*;

if *uplo* = 'L', the lower triangular part of *A*.

Otherwise, all of the matrix *A* is copied.

m — INTEGER. The number of rows in the matrix *A* (*m*≥ 0).

n — INTEGER. The number of columns in *A* (*n*≥ 0).

a — REAL for slacpy

DOUBLE PRECISION for dlacpy

COMPLEX for clacpy

DOUBLE COMPLEX for zlacpy.

Array *a*(*lda*,*), contains the *m*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1,*n*).

If *uplo* = 'U', only the upper triangle or trapezoid is accessed; if *uplo* = 'L', only the lower triangle or trapezoid is accessed.

lda — INTEGER. The leading dimension of *a*; *lda*≥max(1,*m*).

ldb — INTEGER. The leading dimension of the output array *b*; *ldb*≥ max(1, *m*).

Output Parameters

b — REAL for slacpy

DOUBLE PRECISION for dlacpy

COMPLEX for clacpy

DOUBLE COMPLEX for zlacpy.

Array *b*(*ldb*,*), contains the *m*-by-*n* matrix *B*.

The second dimension of *b* must be at least max(1,*n*).

On exit, *B* = *A* in the locations specified by *uplo*.

?ladiv

Performs complex division in real arithmetic, avoiding unnecessary overflow.

Syntax

```
call sladiv( a, b, c, d, p, q )
call dladiv( a, b, c, d, p, q )
res = cladiv( x, y )
res = zladiv( x, y )
```

Include Files

- mkl.fi

Description

The routines `sladiv/dladiv` perform complex division in real arithmetic as

$$p + iq = \frac{a + ib}{c + id}$$

Complex functions `cladiv/zladiv` compute the result as

res = *x*/*y*,

where *x* and *y* are complex. The computation of *x* / *y* will not overflow on an intermediary step unless the results overflows.

The algorithm used is due to [Baudin12].

Input Parameters

a, *b*, *c*, *d*	`REAL` for `sladiv`
	`DOUBLE PRECISION` for `dladiv`
	The scalars *a*, *b*, *c*, and *d* in the above expression (for real flavors only).

x, *y*	`COMPLEX` for `cladiv`
	`DOUBLE COMPLEX` for `zladiv`
	The complex scalars *x* and *y* (for complex flavors only).

Output Parameters

p, *q*	`REAL` for `sladiv`
	`DOUBLE PRECISION` for `dladiv`
	The scalars *p* and *q* in the above expression (for real flavors only).

res	`COMPLEX` for `cladiv`
	`DOUBLE COMPLEX` for `zladiv`
	Contains the result of division *x* / *y*.

?lae2

Computes the eigenvalues of a 2-by-2 symmetric matrix.

Syntax

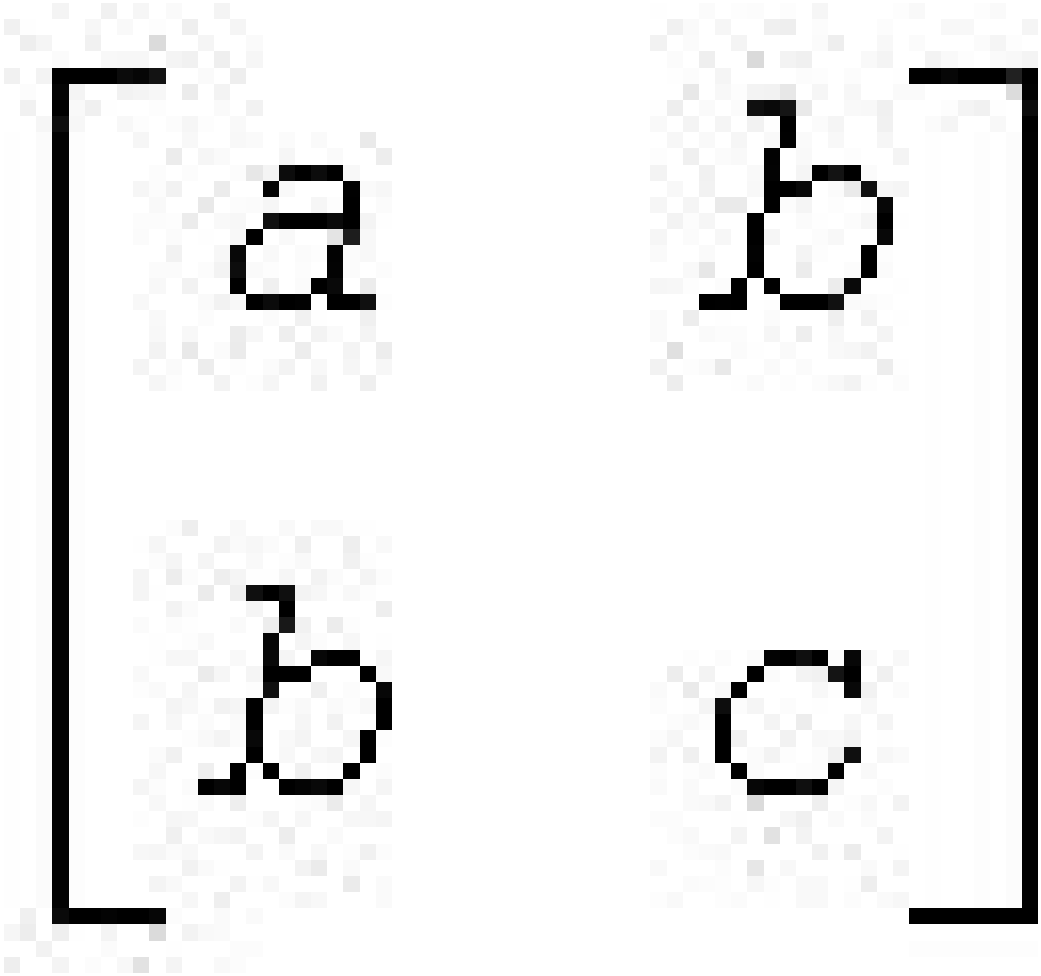
```
call slae2( a, b, c, rt1, rt2 )
call dlae2( a, b, c, rt1, rt2 )
```

Include Files

- mkl.fi

Description

The routines sla2/dlae2 compute the eigenvalues of a 2-by-2 symmetric matrix



On return, *rt1* is the eigenvalue of larger absolute value, and *rt1* is the eigenvalue of smaller absolute value.

Input Parameters

a, *b*, *c*

REAL for slae2

DOUBLE PRECISION for dlae2

The elements *a*, *b*, and *c* of the 2-by-2 matrix above.

Output Parameters

rt1, *rt2*

REAL for slae2

DOUBLE PRECISION for dlae2

The computed eigenvalues of larger and smaller absolute value, respectively.

Application Notes

rt1 is accurate to a few ulps barring over/underflow. *rt2* may be inaccurate if there is massive cancellation in the determinant *a*c-b*b*; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute *rt2* accurately in all cases.

Overflow is possible only if *rt1* is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds

underflow_threshold / `macheps`.

?laebz

Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine `?stebz`.

Syntax

`call slaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol, reltol, pivmin, d, e, e2, nval, ab, c, mout, nab, work, iwork, info )`

`call dlaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol, reltol, pivmin, d, e, e2, nval, ab, c, mout, nab, work, iwork, info )`

Include Files

- `mkl.fi`

Description

The routine `?laebz` contains the iteration loops which compute and use the function *n*(*w*), which is the count of eigenvalues of a symmetric tridiagonal matrix *T* less than or equal to its argument *w*. It performs a choice of two types of loops:

ijob	=1, followed by
ijob	=2: It takes as input a list of intervals and returns a list of sufficiently small intervals whose union contains the same eigenvalues as the union of the original intervals. The input intervals are `(ab(j,1),ab(j,2)], j=1,...,minp`. The output interval `(ab(j,1),ab(j,2)]` will contain eigenvalues `nab(j,1)+1,...,nab(j,2)`, where `1 ≤ j ≤ mout`.
ijob	=3: It performs a binary search in each input interval `(ab(j,1),ab(j,2)]` for a point *w*(*j*) such that `n(w(j))=nval(j)`, and uses *c*(*j*) as the starting point of the search. If such a *w*(*j*) is found, then on output `ab(j,1)=ab(j,2)=w`. If no such *w*(*j*) is found, then on output `(ab(j,1),ab(j,2)]` will be a small interval containing the point where *n*(*w*) jumps through *nval*(*j*), unless that point lies outside the initial interval.

Note that the intervals are in all cases half-open intervals, that is, of the form `(a,b]`, which includes *b* but not *a* .

To avoid underflow, the matrix should be scaled so that its largest element is no greater than $overflow^{1/2} * overflow^{1/4}$ in absolute value. To assure the most accurate computation of small eigenvalues, the matrix should be scaled to be not much smaller than that, either.

NOTE
In general, the arguments are not checked for unreasonable values.

Input Parameters

ijob

INTEGER. Specifies what is to be done:

= 1: Compute *nab* for the initial intervals.

= 2: Perform bisection iteration to find eigenvalues of *T*.

= 3: Perform bisection iteration to invert *n*(*w*), i.e., to find a point which has a specified number of eigenvalues of *T* to its left. Other values will cause ?laebz to return with *info*=-1.

nitmax

INTEGER. The maximum number of "levels" of bisection to be performed, i.e., an interval of width *W* will not be made smaller than $2^{-nitmax*W}$. If not all intervals have converged after *nitmax* iterations, then *info* is set to the number of non-converged intervals.

n

INTEGER. The dimension *n* of the tridiagonal matrix *T*. It must be at least 1.

mmax

INTEGER. The maximum number of intervals. If more than *mmax* intervals are generated, then ?laebz will quit with *info*=*mmax*+1.

minp

INTEGER. The initial number of intervals. It may not be greater than *mmax*.

nbmin

INTEGER. The smallest number of intervals that should be processed using a vector loop. If zero, then only the scalar loop will be used.

abstol

REAL for slaebz

DOUBLE PRECISION for dlaebz.

The minimum (absolute) width of an interval. When an interval is narrower than *abstol*, or than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. This must be at least zero.

reltol

REAL for slaebz

DOUBLE PRECISION for dlaebz.

The minimum relative width of an interval. When an interval is narrower than *abstol*, or than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least *radix***machineepsilon*.

pivmin

REAL for slaebz

DOUBLE PRECISION for dlaebz.

The minimum absolute value of a "pivot" in the Sturm sequence loop. This value **must** be at least (`max |e(j)**2|*safe_min`) and at least *safe_min*, where *safe_min* is at least the smallest number that can divide one without overflow.

d, *e*, *e2*

`REAL` for `slaebz`

`DOUBLE PRECISION` for `dlaebz`.

Arrays, dimension (*n*) each. The array *d* contains the diagonal elements of the tridiagonal matrix *T*.

The array *e* contains the off-diagonal elements of the tridiagonal matrix *T* in positions 1 through *n*-1. *e*(*n*)vis arbitrary.

The array *e2* contains the squares of the off-diagonal elements of the tridiagonal matrix *T*. *e2*(*n*) is ignored.

nval

`INTEGER`.

Array, dimension (*minp*).

If *ijob*=1 or 2, not referenced.

If *ijob*=3, the desired values of *n*(*w*).

ab

`REAL` for `slaebz`

`DOUBLE PRECISION` for `dlaebz`.

Array, dimension (*mmax*,2) The endpoints of the intervals. *ab*(j,1) is *a*(*j*), the left endpoint of the *j*-th interval, and *ab*(j,2) is *b*(*j*), the right endpoint of the *j*-th interval.

c

`REAL` for `slaebz`

`DOUBLE PRECISION` for `dlaebz`.

Array, dimension (*mmax*)

If *ijob*=1, ignored.

If *ijob*=2, workspace.

If *ijob*=3, then on input *c*(*j*) should be initialized to the first search point in the binary search.

nab

`INTEGER`.

Array, dimension (*mmax*,2)

If *ijob*=2, then on input, *nab*(i,j) should be set. It must satisfy the condition:

$n(ab(i,1)) \le nab(i,1) \le nab(i,2) \le n(ab(i,2))$, which means that in interval i only eigenvalues $nab(i,1)+1,\ldots,nab(i,2)$ are considered. Usually, $nab(i,j)=n(ab(i,j))$, from a previous call to `?laebz` with *ijob*=1.

If *ijob*=3, normally, *nab* should be set to some distinctive value(s) before `?laebz` is called.

work

`REAL` for `slaebz`

`DOUBLE PRECISION` for `dlaebz`.

Workspace array, dimension (*mmax*).

iwork INTEGER.

Workspace array, dimension (*mmax*).

Output Parameters

nval The elements of *nval* will be reordered to correspond with the intervals in *ab*. Thus, *nval*(*j*) on output will not, in general be the same as *nval*(*j*) on input, but it will correspond with the interval (*ab*(j,1),*ab*(j,2)] on output.

ab The input intervals will, in general, be modified, split, and reordered by the calculation.

mout INTEGER.

If *ijob*=1, the number of eigenvalues in the intervals.

If *ijob*=2 or 3, the number of intervals output.

If *ijob*=3, *mout* will equal *minp*.

nab If *ijob*=1, then on output *nab*(i,j) will be set to *N*(*ab*(i,j)).

If *ijob*=2, then on output, *nab*(i,j) will contain max(*na*(*k*, min(*nb*(*k*), *N*(*ab*(i,j)))), where *k* is the index of the input interval that the output interval (*ab*(j,1),*ab*(j,2)] came from, and *na*(*k*) and *nb*(*k*) are the input values of *nab*(*k*,1) and *nab*(*k*,2).

If *ijob*=3, then on output, *nab*(i,j) contains *N*(*ab*(i,j)), unless *N*(*w*) > *nval*(i) for all search points *w*, in which case *nab*(i,1) will not be modified, i.e., the output value will be the same as the input value (modulo reorderings, see *nval* and *ab*), or unless *N*(*w*) < *nval*(i) for all search points *w*, in which case *nab*(i,2) will not be modified.

info INTEGER.

If *info* = 0 - all intervals converged

If *info* = 1--*mmax* - the last *info* interval did not converge.

If info = mmax+1 - more than *mmax* intervals were generated

Application Notes

This routine is intended to be called only by other LAPACK routines, thus the interface is less user-friendly. It is intended for two purposes:

(a) finding eigenvalues. In this case, ?laebz should have one or more initial intervals set up in *ab*, and ?laebz should be called with *ijob*=1. This sets up *nab*, and also counts the eigenvalues. Intervals with no eigenvalues would usually be thrown out at this point. Also, if not all the eigenvalues in an interval i are desired, *nab*(i,1) can be increased or *nab*(i,2) decreased. For example, set *nab*(i,1)=*nab*(i,2)-1 to get the largest eigenvalue. ?laebz is then called with *ijob*=2 and *mmax* no smaller than the value of *mout* returned by the call with *ijob*=1. After this (*ijob*=2) call, eigenvalues *nab*(i,1)+1 through *nab*(i,2) are approximately *ab*(i,1) (or *ab*(i,2)) to the tolerance specified by *abstol* and *reltol*.

(b) finding an interval (*a*',*b*'] containing eigenvalues *w*(f),...,*w*(l). In this case, start with a Gershgorin interval (*a*,*b*). Set up *ab* to contain 2 search intervals, both initially (*a*,*b*). One *nval* element should contain f-1 and the other should contain l, while *c* should contain *a* and *b*, respectively. *nab*(i,1) should be -1 and *nab*(i,2) should be *n*+1, to flag an error if the desired interval does not lie in (*a*,*b*). ?laebz is then called with

ijob=3. On exit, if `w(f-1) < w(f)`, then one of the intervals -- j -- will have `ab(j,1)=ab(j,2)` and `nab(j,1)=nab(j,2)=f-1`, while if, to the specified tolerance, `w(f-k)=...=w(f+r), k > 0` and `r ≥ 0`, then the interval will have `n(ab(j,1))=nab(j,1)=f-k` and `n(ab(j,2))=nab(j,2)=f+r`. The cases `w(l) < w(l+1)` and *w*(l-r)=...=*w*(l+k) are handled similarly.

?laed0

Used by `?stedc`. *Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.*

Syntax

```
call slaed0( icompq, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info )
call dlaed0( icompq, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info )
call claed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
call zlaed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
```

Include Files

- `mkl.fi`

Description

Real flavors of this routine compute all eigenvalues and (optionally) corresponding eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Complex flavors `claed0`/`zlaed0` compute all eigenvalues of a symmetric tridiagonal matrix which is one diagonal block of those from reducing a dense or band Hermitian matrix and corresponding eigenvectors of the dense or band matrix.

Input Parameters

icompq

`INTEGER`. Used with real flavors only.

If *icompq* = 0, compute eigenvalues only.

If *icompq* = 1, compute eigenvectors of original dense symmetric matrix also.

On entry, the array *q* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.

If *icompq* = 2, compute eigenvalues and eigenvectors of the tridiagonal matrix.

qsiz

`INTEGER`.

The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; *qsiz*≥*n* (for real flavors, *qsiz*≥*n* if *icompq* = 1).

n

`INTEGER`. The dimension of the symmetric tridiagonal matrix (*n*≥ 0).

d, *e*, *rwork*

`REAL` for single-precision flavors

`DOUBLE PRECISION` for double-precision flavors. Arrays:

d(*) contains the main diagonal of the tridiagonal matrix. The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of the tridiagonal matrix. The dimension of *e* must be at least max(1, *n*–1).

rwork(*) is a workspace array used in complex flavors only. The dimension of *rwork* must be at least $(1 + 3n + 2n\log_2(n) + 3n^2)$, where $\log_2(n)$ = smallest integer k such that $2^k \geq n$.

q, *qstore*

REAL for slaed0

DOUBLE PRECISION for dlaed0

COMPLEX for claed0

DOUBLE COMPLEX for zlaed0.

Arrays: *q*(*ldq*, *), *qstore*(*ldqs*, *). The second dimension of these arrays must be at least max(1, *n*).

For real flavors:

If *icompq* = 0, array *q* is not referenced.

If *icompq* = 1, on entry, *q* is a subset of the columns of the orthogonal matrix used to reduce the full matrix to tridiagonal form corresponding to the subset of the full matrix which is being decomposed at this time.

If *icompq* = 2, on entry, *q* will be the identity matrix. The array *qstore* is a workspace array referenced only when *icompq* = 1. Used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

For complex flavors:

On entry, *q* must contain an *qsiz*-by-*n* matrix whose columns are unitarily orthonormal. It is a part of the unitary matrix that reduces the full dense Hermitian matrix to a (reducible) symmetric tridiagonal matrix. The array *qstore* is a workspace array used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

ldq

INTEGER. The leading dimension of the array *q*; *ldq* ≥ max(1, *n*).

ldqs

INTEGER. The leading dimension of the array *qstore*; *ldqs* ≥ max(1, *n*).

work

REAL for slaed0

DOUBLE PRECISION for dlaed0.

Workspace array, used in real flavors only.

If *icompq* = 0 or 1, the dimension of *work* must be at least $(1 + 3n + 2n\log_2(n) + 3n^2)$, where $\log_2(n)$ = smallest integer k such that $2^k \geq n$.

If *icompq* = 2, the dimension of *work* must be at least $(4n + n^2)$.

iwork

INTEGER.

Workspace array.

For real flavors, if *icompq* = 0 or 1, and for complex flavors, the dimension of *iwork* must be at least $(6 + 6n + 5n\log_2(n))$.

For real flavors, if *icompq* = 2, the dimension of *iwork* must be at least (3+5*n*).

Output Parameters

d — On exit, contains eigenvalues in ascending order.

e — On exit, the array is destroyed.

q — If *icompq* = 2, on exit, *q* contains the eigenvectors of the tridiagonal matrix.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i* >0, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *i*/(*n*+1) through mod(*i*, *n*+1).

?laed1

Used by sstedc/dstedc*. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.*

Syntax

```
call slaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
call dlaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
```

Include Files

- mkl.fi

Description

The routine ?laed1 computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and eigenvectors of a tridiagonal matrix. ?laed7 handles the case in which eigenvalues only or eigenvalues and eigenvectors of a full symmetric matrix (which was reduced to tridiagonal form) are desired.

$T = Q(\text{in})*(D(\text{in})+ rho*Z*Z^T)*Q^T(\text{in}) = Q(\text{out})*D(\text{out})*Q^T(\text{out})$

where $Z = Q^T u$, u is a vector of length *n* with ones in the *cutpnt* and (*cutpnt*+1) -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in *Q*, and the eigenvalues are in *D*. The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the *z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?laed2.

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine ?laed4 (as called by ?laed3). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

n	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
d, *q*, *work*	REAL for slaed1 DOUBLE PRECISION for dlaed1. Arrays: *d*(*) contains the eigenvalues of the rank-1-perturbed matrix. The dimension of *d* must be at least max(1, *n*). *q*(*ldq*, *) contains the eigenvectors of the rank-1-perturbed matrix. The second dimension of *q* must be at least max(1, *n*). *work*(*) is a workspace array, dimension at least ($4n+n^2$).
ldq	INTEGER. The leading dimension of the array *q*; $ldq \geq \max(1, n)$.
indxq	INTEGER. Array, dimension (*n*). On entry, the permutation which separately sorts the two subproblems in *d* into ascending order.
rho	REAL for slaed1 DOUBLE PRECISION for dlaed1. The subdiagonal entry used to create the rank-1 modification. This parameter can be modified by ?laed2, where it is input/output.
cutpnt	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1,n) \leq cutpnt \leq n/2$.
iwork	INTEGER. Workspace array, dimension (4*n*).

Output Parameters

d	On exit, contains the eigenvalues of the repaired matrix.
q	On exit, *q* contains the eigenvectors of the repaired tridiagonal matrix.
indxq	On exit, contains the permutation which will reintegrate the subproblems back into sorted order, that is, *d*(*indxq*(i = 1, *n*)) will be in ascending order.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, an eigenvalue did not converge.

?laed2

Used by `sstedc/dstedc`*. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.*

Syntax

```
call slaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc, indxp, coltyp, info )
call dlaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc, indxp, coltyp, info )
```

Include Files

- `mkl.fi`

Description

The routine `?laed2` merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny entry in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

k	`INTEGER`. The number of non-deflated eigenvalues, and the order of the related secular equation (`0 ≤ k ≤ n`).
n	`INTEGER`. The dimension of the symmetric tridiagonal matrix (`n≥ 0`).
n1	`INTEGER`. The location of the last eigenvalue in the leading sub-matrix; `min(1,n) ≤ n1 ≤ n/2`.
d, *q*, *z*	`REAL` for `slaed2` `DOUBLE PRECISION` for `dlaed2`. Arrays: `d(*)` contains the eigenvalues of the two submatrices to be combined. The dimension of *d* must be at least `max(1, n)`. `q(ldq, *)` contains the eigenvectors of the two submatrices in the two square blocks with corners at (1,1), (*n1*,*n1*) and (*n1*+1,*n1*+1), (*n*,*n*). The second dimension of *q* must be at least `max(1, n)`. `z(*)` contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).
ldq	`INTEGER`. The leading dimension of the array *q*; `ldq≥ max(1, n)`.
indxq	`INTEGER`. Array, dimension (*n*). On entry, the permutation which separately sorts the two subproblems in *d* into ascending order. Note that elements in the second half of this permutation must first have *n1* added to their values.
rho	`REAL` for `slaed2`

DOUBLE PRECISION for dlaed2.

On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.

indx, *indxp*

INTEGER.

Workspace arrays, dimension (*n*) each. Array *indx* contains the permutation used to sort the contents of *dlamda* into ascending order.

Array *indxp* contains the permutation used to place deflated values of *d* at the end of the array.

indxp(1:*k*) points to the nondeflated *d*-values and *indxp*(*k*+1:*n*) points to the deflated eigenvalues.

coltyp

INTEGER.

Workspace array, dimension (*n*).

During execution, a label which will indicate which of the following types a column in the *q2* matrix is:

1 : non-zero in the upper half only;

2 : dense;

3 : non-zero in the lower half only;

4 : deflated.

Output Parameters

d

On exit, *d* contains the trailing (*n*-*k*) updated eigenvalues (those which were deflated) sorted into increasing order.

q

On exit, *q* contains the trailing (*n*-*k*) updated eigenvectors (those which were deflated) in its last *n*-*k* columns.

z

On exit, *z* content is destroyed by the updating process.

indxq

Destroyed on exit.

rho

On exit, *rho* has been modified to the value required by ?laed3.

dlamda, *w*, *q2*

REAL for slaed2

DOUBLE PRECISION for dlaed2.

Arrays: *dlamda*(*n*), *w*(*n*), *q2*($n1^2$+(*n*-*n1*)^2).

The array *dlamda* contains a copy of the first *k* eigenvalues which is used by ?laed3 to form the secular equation.

The array *w* contains the first *k* values of the final deflation-altered *z*-vector which is passed to ?laed3.

The array *q2* contains a copy of the first *k* eigenvectors which is used by ?laed3 in a matrix multiply (sgemm/dgemm) to solve for the new eigenvectors.

indxc

INTEGER. Array, dimension (*n*).

The permutation used to arrange the columns of the deflated *q* matrix into three groups: the first group contains non-zero elements only at and above *n1*, the second contains non-zero elements only below *n1*, and the third is dense.

coltyp — On exit, *coltyp*(i) is the number of columns of type i, for i=1 to 4 only (see the definition of types in the description of *coltyp* in *Input Parameters*).

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?laed3

Used by sstedc/dstedc*. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.*

Syntax

```
call slaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
call dlaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
```

Include Files

- mkl.fi

Description

The routine ?laed3 finds the roots of the secular equation, as defined by the values in *d*, *w*, and *rho*, between 1 and *k*.

It makes the appropriate calls to ?laed4 and then updates the eigenvectors by multiplying the matrix of eigenvectors of the pair of eigensystems being combined by the matrix of eigenvectors of the *k*-by-*k* system which is solved here.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but none are known.

Input Parameters

k — INTEGER. The number of terms in the rational function to be solved by ?laed4 (*k*≥ 0).

n — INTEGER. The number of rows and columns in the *q* matrix. *n*≥*k* (deflation may result in *n* >*k*).

n1 — INTEGER. The location of the last eigenvalue in the leading sub-matrix; min(1,*n*) ≤ *n1* ≤ *n*/2.

q — REAL for slaed3

DOUBLE PRECISION for dlaed3.

Array *q*(*ldq*, *). The second dimension of *q* must be at least max(1, *n*).

Initially, the first *k* columns of this array are used as workspace.

ldq INTEGER. The leading dimension of the array *q*; *ldq*≥ max(1, *n*).

rho REAL for slaed3

DOUBLE PRECISION for dlaed3.

The value of the parameter in the rank one update equation. *rho*≥ 0 required.

dlamda, *q2*, *w* REAL for slaed3

DOUBLE PRECISION for dlaed3.

Arrays: *dlamda*(*k*), *q2*(*ldq2*, *), *w*(*k*).

The first *k* elements of the array *dlamda* contain the old roots of the deflated updating problem. These are the poles of the secular equation.

The first *k* columns of the array *q2* contain the non-deflated eigenvectors for the split problem. The second dimension of *q2* must be at least max(1, *n*).

The first *k* elements of the array *w* contain the components of the deflation-adjusted updating vector.

indx INTEGER. Array, dimension (*n*).

The permutation used to arrange the columns of the deflated *q* matrix into three groups (see ?laed2).

The rows of the eigenvectors found by ?laed4 must be likewise permuted before the matrix multiply can take place.

ctot INTEGER. Array, dimension (4).

A count of the total number of the various types of columns in *q*, as described in *indx*. The fourth column type is any column which has been deflated.

s REAL for slaed3

DOUBLE PRECISION for dlaed3.

Workspace array, dimension (*n1*+1)**k* .

Will contain the eigenvectors of the repaired matrix which will be multiplied by the previously accumulated eigenvectors to update the system.

Output Parameters

d REAL for slaed3

DOUBLE PRECISION for dlaed3.

Array, dimension at least max(1, *n*).

d(i) contains the updated eigenvalues for 1 ≤ i ≤ *k*.

q On exit, the columns 1 to *k* of *q* contain the updated eigenvectors.

dlamda May be changed on output by having lowest order bit set to zero on Cray X-MP, Cray Y-MP, Cray-2, or Cray C-90, as described above.

w	Destroyed on exit.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, an eigenvalue did not converge.

?laed4

Used by sstedc/dstedc*. Finds a single root of the secular equation.*

Syntax

```
call slaed4( n, i, d, z, delta, rho, dlam, info )
call dlaed4( n, i, d, z, delta, rho, dlam, info )
```

Include Files

- mkl.fi

Description

This routine computes the *i*-th updated eigenvalue of a symmetric rank-one modification to a diagonal matrix whose elements are given in the array *d*, and that

D(i) < *D*(*j*) for *i* < *j*

and that *rho* > 0. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

diag(*D*) + *rho***Z* * transpose(*Z*).

where we assume the Euclidean norm of *Z* is 1.

The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

n	INTEGER. The length of all arrays.
i	INTEGER. The index of the eigenvalue to be computed; 1 ≤ i ≤ *n*.
d, *z*	REAL for slaed4 DOUBLE PRECISION for dlaed4 Arrays, dimension (*n*) each. The array *d* contains the original eigenvalues. It is assumed that they are in order, *d*(i) < *d*(*j*) for *i* < *j*. The array *z* contains the components of the updating vector *Z*.
rho	REAL for slaed4 DOUBLE PRECISION for dlaed4 The scalar in the symmetric updating formula.

Output Parameters

delta — REAL for slaed4

DOUBLE PRECISION for dlaed4

Array, dimension (*n*).

If *n* ≠ 1, *delta* contains (*d(j) – lambda_i*) in its *j*-th component. If *n* = 1, then *delta*(1) = 1. The vector *delta* contains the information necessary to construct the eigenvectors.

dlam — REAL for slaed4

DOUBLE PRECISION for dlaed4

The computed *lambda_i*, the *i*-th updated eigenvalue.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, the updating process failed.

?laed5

Used by sstedc/dstedc*. Solves the 2-by-2 secular equation.*

Syntax

```
call slaed5( i, d, z, delta, rho, dlam )
call dlaed5( i, d, z, delta, rho, dlam )
```

Include Files

- mkl.fi

Description

The routine computes the *i*-th eigenvalue of a symmetric rank-one modification of a 2-by-2 diagonal matrix diag(*D*) + *rho***Z* * transpose(*Z*).

The diagonal elements in the array *D* are assumed to satisfy

D(i) < *D*(*j*) for *i* < *j*.

We also assume *rho* > 0 and that the Euclidean norm of the vector *Z* is one.

Input Parameters

i — INTEGER. The index of the eigenvalue to be computed;

1 ≤ i ≤ 2.

d, *z* — REAL for slaed5

DOUBLE PRECISION for dlaed5

Arrays, dimension (2) each. The array *d* contains the original eigenvalues. It is assumed that *d*(1) < *d*(2).

The array *z* contains the components of the updating vector.

rho

`REAL` for `slaed5`

`DOUBLE PRECISION` for `dlaed5`

The scalar in the symmetric updating formula.

Output Parameters

delta

`REAL` for `slaed5`

`DOUBLE PRECISION` for `dlaed5`

Array, dimension (2).

The vector *delta* contains the information necessary to construct the eigenvectors.

dlam

`REAL` for `slaed5`

`DOUBLE PRECISION` for `dlaed5`

The computed *lambda_i*, the *i*-th updated eigenvalue.

?laed6

Used by `sstedc`/`dstedc`*. Computes one Newton step in solution of the secular equation.*

Syntax

```
call slaed6( kniter, orgati, rho, d, z, finit, tau, info )
call dlaed6( kniter, orgati, rho, d, z, finit, tau, info )
```

Include Files

- `mkl.fi`

Description

The routine computes the positive or negative root (closest to the origin) of

$$f(x) = rho + \frac{z(1)}{d(1) - x} + \frac{z(2)}{d(2) - x} + \frac{z(3)}{d(3) - x}$$

It is assumed that if *orgati* = `.TRUE.` the root is between *d*(2) and *d*(3);otherwise it is between *d*(1) and *d*(2) This routine is called by `?laed4` when necessary. In most cases, the root sought is the smallest in magnitude, though it might not be in some extremely rare situations.

Input Parameters

kniter

`INTEGER`.

Refer to `?laed4` for its significance.

orgati

`LOGICAL`.

If *orgati* = .TRUE., the needed root is between *d*(2) and *d*(3); otherwise it is between *d*(1) and *d*(2). See ?laed4 for further details.

rho

REAL for slaed6

DOUBLE PRECISION for dlaed6

Refer to the equation for *f*(*x*) above.

d, *z*

REAL for slaed6

DOUBLE PRECISION for dlaed6

Arrays, dimension (3) each.

The array *d* satisfies *d*(1) < *d*(2) < *d*(3).

Each of the elements in the array *z* must be positive.

finit

REAL for slaed6

DOUBLE PRECISION for dlaed6

The value of *f*(*x*) at 0. It is more accurate than the one evaluated inside this routine (if someone wants to do so).

Output Parameters

tau

REAL for slaed6

DOUBLE PRECISION for dlaed6

The root of the equation for *f*(*x*).

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, failure to converge.

?laed7

Used by ?stedc*. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.*

Syntax

```
call slaed7( icompq, n, qsiz, tlvls, curlvl, curpbm, d, q, ldq, indxq, rho, cutpnt,
qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info )
```

```
call dlaed7( icompq, n, qsiz, tlvls, curlvl, curpbm, d, q, ldq, indxq, rho, cutpnt,
qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info )
```

```
call claed7( n, cutpnt, qsiz, tlvls, curlvl, curpbm, d, q, ldq, rho, indxq, qstore,
qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info )
```

```
call zlaed7( n, cutpnt, qsiz, tlvls, curlvl, curpbm, d, q, ldq, rho, indxq, qstore,
qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info )
```

Include Files

- mkl.fi

Description

The routine ?laed7 computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and optionally eigenvectors of a dense symmetric/Hermitian matrix that has been reduced to tridiagonal form. For real flavors, slaed1/dlaed1 handles the case in which all eigenvalues and eigenvectors of a symmetric tridiagonal matrix are desired.

$T = Q(\text{in})*(D(\text{in})+rho*Z*Z^T)*Q^T(\text{in}) = Q(\text{out})*D(\text{out})*Q^T(\text{out})$ for real flavors, or

$T = Q(\text{in})*(D(\text{in})+rho*Z*Z^H)*Q^H(\text{in}) = Q(\text{out})*D(\text{out})*Q^H(\text{out})$ for complex flavors

where $Z = Q^T*u$ for real flavors and $Z = Q^H*u$ for complex flavors, *u* is a vector of length *n* with ones in the *cutpnt* and ($cutpnt + 1$) -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in *Q*, and the eigenvalues are in *D*. The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the *z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine slaed8/dlaed8 (for real flavors) or by the routine slaed2/dlaed2 (for complex flavors).

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine ?laed4 (as called by ?laed9 or ?laed3). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

icompq
INTEGER. Used with real flavors only.
If $icompq = 0$, compute eigenvalues only.
If $icompq = 1$, compute eigenvectors of original dense symmetric matrix also. On entry, the array *q* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.

n
INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).

cutpnt
INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1,n) \leq cutpnt \leq n$.

qsiz
INTEGER.
The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if $icompq = 1$).

tlvls
INTEGER. The total number of merging levels in the overall divide and conquer tree.

curlvl
INTEGER. The current level in the overall merge routine, $0 \leq curlvl \leq tlvls$.

curpbm
INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).

d
REAL for slaed7/claed7

DOUBLE PRECISION for dlaed7/zlaed7.

Array, dimension at least max(1, *n*).

Array *d*(*) contains the eigenvalues of the rank-1-perturbed matrix.

q, *work*

REAL for slaed7

DOUBLE PRECISION for dlaed7

COMPLEX for claed7

DOUBLE COMPLEX for zlaed7.

Arrays:

q(*ldq*, *) contains the eigenvectors of the rank-1-perturbed matrix. The second dimension of *q* must be at least max(1, *n*).

work(*) is a workspace array, dimension at least (3*n*+2**qsiz***n*) for real flavors and at least (*qsiz***n*) for complex flavors.

ldq

INTEGER. The leading dimension of the array *q*; *ldq*≥ max(1, *n*).

indxq

INTEGER. Array, dimension (*n*).

Contains the permutation that separately sorts the two sub-problems in *d* into ascending order.

rho

REAL for slaed7 /claed7

DOUBLE PRECISION for dlaed7/zlaed7.

The subdiagonal element used to create the rank-1 modification.

qstore

REAL for slaed7/claed7

DOUBLE PRECISION for dlaed7/zlaed7.

Array, dimension (n^2+1). Serves also as output parameter.

Stores eigenvectors of submatrices encountered during divide and conquer, packed together. *qptr* points to beginning of the submatrices.

qptr

INTEGER. Array, dimension (*n*+2). Serves also as output parameter. List of indices pointing to beginning of submatrices stored in *qstore*. The submatrices are numbered starting at the bottom left of the divide and conquer tree, from left to right and bottom to top.

prmptr, *perm*, *givptr*

INTEGER. Arrays, dimension ($n \log_2 n$) each.

The array *prmptr*(*) contains a list of pointers which indicate where in *perm* a level's permutation is stored. *prmptr*(i+1) – *prmptr*(i) indicates the size of the permutation and also the size of the full, non-deflated problem.

The array *perm*(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock. This parameter can be modified by ?laed8, where it is output.

The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. *givptr*(i+1) – *givptr*(i) indicates the number of Givens rotations.

givcol

INTEGER. Array, dimension (2, $n \log_2 n$).

Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

givnum REAL for slaed7/claed7

DOUBLE PRECISION for dlaed7/zlaed7.

Array, dimension (2, $n \log_2 n$).

Each number indicates the *S* value to be used in the corresponding Givens rotation.

iwork INTEGER.

Workspace array, dimension (4*n*).

rwork REAL for claed7

DOUBLE PRECISION for zlaed7.

Workspace array, dimension (3*n*+2*qsiz***n*). Used in complex flavors only.

Output Parameters

d On exit, contains the eigenvalues of the repaired matrix.

q On exit, *q* contains the eigenvectors of the repaired tridiagonal matrix.

indxq INTEGER. Array, dimension (*n*).

Contains the permutation that reintegrates the subproblems back into a sorted order, that is,

d(*indxq*(i = 1, *n*)) will be in the ascending order.

rho This parameter can be modified by ?laed8, where it is input/output.

prmptr, *perm*, *givptr* INTEGER. Arrays, dimension ($n \log_2 n$) each.

The array *prmptr* contains an updated list of pointers.

The array *perm* contains an updated permutation.

The array *givptr* contains an updated list of pointers.

givcol This parameter can be modified by ?laed8, where it is output.

givnum This parameter can be modified by ?laed8, where it is output.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, an eigenvalue did not converge.

?laed8

Used by ?stedc*. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.*

Syntax

call slaed8(*icompq*, *k*, *n*, *qsiz*, *d*, *q*, *ldq*, *indxq*, *rho*, *cutpnt*, *z*, *dlamda*, *q2*, *ldq2*, *w*, *perm*, *givptr*, *givcol*, *givnum*, *indxp*, *indx*, *info*)

call dlaed8(*icompq*, *k*, *n*, *qsiz*, *d*, *q*, *ldq*, *indxq*, *rho*, *cutpnt*, *z*, *dlamda*, *q2*, *ldq2*, *w*, *perm*, *givptr*, *givcol*, *givnum*, *indxp*, *indx*, *info*)

call claed8(*k*, *n*, *qsiz*, *q*, *ldq*, *d*, *rho*, *cutpnt*, *z*, *dlamda*, *q2*, *ldq2*, *w*, *indxp*, *indx*, *indxq*, *perm*, *givptr*, *givcol*, *givnum*, *info*)

call zlaed8(*k*, *n*, *qsiz*, *q*, *ldq*, *d*, *rho*, *cutpnt*, *z*, *dlamda*, *q2*, *ldq2*, *w*, *indxp*, *indx*, *indxq*, *perm*, *givptr*, *givcol*, *givnum*, *info*)

Include Files

- mkl.fi

Description

The routine merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny element in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

icompq

INTEGER. Used with real flavors only.

If *icompq* = 0, compute eigenvalues only.

If *icompq* = 1, compute eigenvectors of original dense symmetric matrix also.

On entry, the array *q* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.

n

INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).

cutpnt

INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1,n) \leq cutpnt \leq n$.

qsiz

INTEGER.

The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if $icompq = 1$).

d, *z*

REAL for slaed8/claed8

DOUBLE PRECISION for dlaed8/zlaed8.

Arrays, dimension at least max(1, *n*) each. The array *d*(*) contains the eigenvalues of the two submatrices to be combined.

On entry, *z*(*) contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix). The contents of *z* are destroyed by the updating process.

q

REAL for slaed8

DOUBLE PRECISION for dlaed8

COMPLEX for claed8

DOUBLE COMPLEX for zlaed8.

Array

q(*ldq*, *). The second dimension of *q* must be at least max(1, *n*). On entry, *q* contains the eigenvectors of the partially solved system which has been previously updated in matrix multiplies with other partially solved eigensystems.

For real flavors, If *icompq* = 0, *q* is not referenced.

ldq — INTEGER. The leading dimension of the array *q*; *ldq* ≥ max(1, *n*).

ldq2 — INTEGER. The leading dimension of the output array *q2*; *ldq2* ≥ max(1, *n*).

indxq — INTEGER. Array, dimension (*n*).

The permutation that separately sorts the two sub-problems in *d* into ascending order. Note that elements in the second half of this permutation must first have *cutpnt* added to their values in order to be accurate.

rho — REAL for slaed8/claed8

DOUBLE PRECISION for dlaed8/zlaed8.

On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.

Output Parameters

k — INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation.

d — On exit, contains the trailing (*n*-*k*) updated eigenvalues (those which were deflated) sorted into increasing order.

z — On exit, the updating process destroys the contents of *z*.

q — On exit, *q* contains the trailing (*n*-*k*) updated eigenvectors (those which were deflated) in its last (*n*-*k*) columns.

indxq — INTEGER. Array, dimension (*n*).

The permutation of merged eigenvalues set.

rho — On exit, *rho* has been modified to the value required by ?laed3.

dlamda, *w* — REAL for slaed8/claed8

DOUBLE PRECISION for dlaed8/zlaed8.

Arrays, dimension (*n*) each. The array *dlamda*(*) contains a copy of the first *k* eigenvalues which will be used by ?laed3 to form the secular equation.

The array *w*(*) will hold the first *k* values of the final deflation-altered *z*-vector and will be passed to ?laed3.

q2 — REAL for slaed8

DOUBLE PRECISION for dlaed8

COMPLEX for claed8

DOUBLE COMPLEX for zlaed8.

Array

q2(*ldq2*, *). The second dimension of *q2* must be at least max(1, *n*).

Contains a copy of the first *k* eigenvectors which will be used by slaed7/dlaed7 in a matrix multiply (sgemm/dgemm) to update the new eigenvectors. For real flavors, If *icompq* = 0, *q2* is not referenced.

indxp, *indx* — INTEGER. Workspace arrays, dimension (*n*) each.

The array *indxp*(*) will contain the permutation used to place deflated values of *d* at the end of the array. On output, *indxp*(1:*k*) points to the nondeflated *d*-values and *indxp*(*k*+1:*n*) points to the deflated eigenvalues.

The array *indx*(*) will contain the permutation used to sort the contents of *d* into ascending order.

perm — INTEGER. Array, dimension (*n*).

Contains the permutations (from deflation and sorting) to be applied to each eigenblock.

givptr — INTEGER. Contains the number of Givens rotations which took place in this subproblem.

givcol — INTEGER. Array, dimension (2, *n*).

Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

givnum — REAL for slaed8/claed8

DOUBLE PRECISION for dlaed8/zlaed8.

Array, dimension (2, *n*).

Each number indicates the *S* value to be used in the corresponding Givens rotation.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?laed9

Used by sstedc/dstedc*. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.*

Syntax

```
call slaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
call dlaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
```

Include Files

- mkl.fi

Description

The routine finds the roots of the secular equation, as defined by the values in *d*, *z*, and *rho*, between *kstart* and *kstop*. It makes the appropriate calls to slaed4/dlaed4 and then stores the new matrix of eigenvectors for use in calculating the next level of *z* vectors.

Input Parameters

k

INTEGER. The number of terms in the rational function to be solved by slaed4/dlaed4 (*k*≥ 0).

kstart, *kstop*

INTEGER. The updated eigenvalues *lambda*(i),

kstart ≤ i ≤ *kstop* are to be computed.

1 ≤ *kstart* ≤ *kstop* ≤ *k*.

n

INTEGER. The number of rows and columns in the *Q* matrix. *n*≥*k* (deflation may result in *n* > *k*).

q

REAL for slaed9

DOUBLE PRECISION for dlaed9.

Workspace array, dimension (*ldq*, *).

The second dimension of *q* must be at least max(1, *n*).

ldq

INTEGER. The leading dimension of the array *q*; *ldq*≥ max(1, *n*).

rho

REAL for slaed9

DOUBLE PRECISION for dlaed9

The value of the parameter in the rank one update equation. *rho*≥ 0 required.

dlamda, *w*

REAL for slaed9

DOUBLE PRECISION for dlaed9

Arrays, dimension (*k*) each. The first *k* elements of the array *dlamda*(*) contain the old roots of the deflated updating problem. These are the poles of the secular equation.

The first *k* elements of the array *w*(*) contain the components of the deflation-adjusted updating vector.

lds

INTEGER. The leading dimension of the output array *s*; *lds*≥ max(1, *k*).

Output Parameters

d

REAL for slaed9

DOUBLE PRECISION for dlaed9

	Array, dimension (*n*). Elements in *d(i)* are not referenced for `1 ≤ i < kstart` or `kstop < i ≤ n`.
s	`REAL` for `slaed9` `DOUBLE PRECISION` for `dlaed9`. Array, dimension (*lds*, *) . The second dimension of *s* must be at least `max(1, k)`. Will contain the eigenvectors of the repaired matrix which will be stored for subsequent *z* vector calculation and multiplied by the previously accumulated eigenvectors to update the system.
dlamda	On exit, the value is modified to make sure all `dlamda(i) - dlamda(j)` can be computed with high relative accuracy, barring overflow and underflow.
w	Destroyed on exit.
info	`INTEGER`. If `info = 0`, the execution is successful. If `info = -i`, the *i*-th parameter had an illegal value. If `info = 1`, the eigenvalue did not converge.

?laeda

Used by `?stedc`*. Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.*

Syntax

`call slaeda( n, tlvls, curlvl, curpbm, prmptr, perm, givptr, givcol, givnum, q, qptr, z, ztemp, info )`

`call dlaeda( n, tlvls, curlvl, curpbm, prmptr, perm, givptr, givcol, givnum, q, qptr, z, ztemp, info )`

Include Files

- `mkl.fi`

Description

The routine `?laeda` computes the *Z* vector corresponding to the merge step in the *curlvl*-th step of the merge process with *tlvls* steps for the *curpbm*-th problem.

Input Parameters

n	`INTEGER`. The dimension of the symmetric tridiagonal matrix (*n*≥ `0`).
tlvls	`INTEGER`. The total number of merging levels in the overall divide and conquer tree.
curlvl	`INTEGER`. The current level in the overall merge routine, `0 ≤ curlvl ≤ tlvls` .

curpbm — INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).

prmptr, *perm*, *givptr* — INTEGER. Arrays, dimension ($n \log_2 n$) each.

The array *prmptr*(*) contains a list of pointers which indicate where in *perm* a level's permutation is stored. *prmptr*(i+1) – *prmptr*(i) indicates the size of the permutation and also the size of the full, non-deflated problem.

The array *perm*(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock.

The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. *givptr*(i+1) – *givptr*(i) indicates the number of Givens rotations.

givcol — INTEGER. Array, dimension (2, $n \log_2 n$).

Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

givnum — REAL for slaeda

DOUBLE PRECISION for dlaeda.

Array, dimension (2, $n \log_2 n$).

Each number indicates the *S* value to be used in the corresponding Givens rotation.

q — REAL for slaeda

DOUBLE PRECISION for dlaeda.

Array, dimension (n^2).

Contains the square eigenblocks from previous levels, the starting positions for blocks are given by *qptr*.

qptr — INTEGER. Array, dimension (*n*+2). Contains a list of pointers which indicate where in *q* an eigenblock is stored. sqrt(*qptr*(i+1) – *qptr*(i)) indicates the size of the block.

ztemp — REAL for slaeda

DOUBLE PRECISION for dlaeda.

Workspace array, dimension (*n*).

Output Parameters

z — REAL for slaeda

DOUBLE PRECISION for dlaeda.

Array, dimension (*n*). Contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*-th parameter had an illegal value.

?laein

Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.

Syntax

```
call slaein( rightv, noinit, n, h, ldh, wr, wi, vr, vi, b, ldb, work, eps3, smlnum, bignum, info )
call dlaein( rightv, noinit, n, h, ldh, wr, wi, vr, vi, b, ldb, work, eps3, smlnum, bignum, info )
call claein( rightv, noinit, n, h, ldh, w, v, b, ldb, rwork, eps3, smlnum, info )
call zlaein( rightv, noinit, n, h, ldh, w, v, b, ldb, rwork, eps3, smlnum, info )
```

Include Files

- mkl.fi

Description

The routine ?laein uses inverse iteration to find a right or left eigenvector corresponding to the eigenvalue (*wr*,*wi*) of a real upper Hessenberg matrix *H* (for real flavors slaein/dlaein) or to the eigenvalue *w* of a complex upper Hessenberg matrix *H* (for complex flavors claein/zlaein).

Input Parameters

rightv

LOGICAL.

If *rightv* = .TRUE., compute right eigenvector;

if *rightv* = .FALSE., compute left eigenvector.

noinit

LOGICAL.

If *noinit* = .TRUE., no initial vector is supplied in (*vr*,*vi*) or in *v* (for complex flavors);

if *noinit* = .FALSE., initial vector is supplied in (*vr*,*vi*) or in *v* (for complex flavors).

n

INTEGER. The order of the matrix *H* ($n \geq 0$).

h

REAL for slaein

DOUBLE PRECISION for dlaein

COMPLEX for claein

DOUBLE COMPLEX for zlaein.

Array *h*(*ldh*, *).

The second dimension of *h* must be at least max(1, *n*). Contains the upper Hessenberg matrix *H*.

ldh

INTEGER. The leading dimension of the array *h*; $ldh \geq \max(1, n)$.

wr, *wi*

REAL for slaein

DOUBLE PRECISION for dlaein.

The real and imaginary parts of the eigenvalue of *H* whose corresponding right or left eigenvector is to be computed (for real flavors of the routine).

w

COMPLEX for claein

DOUBLE COMPLEX for zlaein.

The eigenvalue of *H* whose corresponding right or left eigenvector is to be computed (for complex flavors of the routine).

vr, *vi*

REAL for slaein

DOUBLE PRECISION for dlaein.

Arrays, dimension (*n*) each. Used for real flavors only. On entry, if *noinit* = .FALSE. and *wi* = 0.0, *vr* must contain a real starting vector for inverse iteration using the real eigenvalue *wr*;

if *noinit* = .FALSE. and *wi*≠ 0.0, *vr* and *vi* must contain the real and imaginary parts of a complex starting vector for inverse iteration using the complex eigenvalue (*wr*,*wi*);otherwise *vr* and *vi* need not be set.

v

COMPLEX for claein

DOUBLE COMPLEX for zlaein.

Array, dimension (*n*). Used for complex flavors only. On entry, if *noinit* = .FALSE., *v* must contain a starting vector for inverse iteration; otherwise *v* need not be set.

b

REAL for slaein

DOUBLE PRECISION for dlaein

COMPLEX for claein

DOUBLE COMPLEX for zlaein.

Workspace array *b*(*ldb*, *). The second dimension of *b* must be at least max(1, *n*).

ldb

INTEGER. The leading dimension of the array *b*;

ldb≥*n*+1 for real flavors;

ldb≥ max(1, *n*) for complex flavors.

work

REAL for slaein

DOUBLE PRECISION for dlaein.

Workspace array, dimension (*n*).

Used for real flavors only.

rwork

REAL for claein

DOUBLE PRECISION for zlaein.

Workspace array, dimension (*n*).

Used for complex flavors only.

eps3, *smlnum*

REAL for slaein/claein

DOUBLE PRECISION for dlaein/zlaein.

eps3 is a small machine-dependent value which is used to perturb close eigenvalues, and to replace zero pivots.

smlnum is a machine-dependent value close to underflow threshold. A suggested value for *smlnum* is `slamch('s') * (n/slamch('p')` for `slaein/claein` or `dlamch('s') * (n/dlamch('p')` for `dlaein/zlaein`. See lamch.

bignum

`REAL` for `slaein`

`DOUBLE PRECISION` for `dlaein`.

bignum is a machine-dependent value close to overflow threshold. Used for real flavors only. A suggested value for *bignum* is `1 / slamch('s')` for `slaein/claein` or `1 / dlamch('s')` for `dlaein/zlaein`.

Output Parameters

vr, *vi*

On exit, if *wi* = 0.0 (real eigenvalue), *vr* contains the computed real eigenvector; if *wi* ≠ 0.0 (complex eigenvalue), *vr* and *vi* contain the real and imaginary parts of the computed complex eigenvector. The eigenvector is normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (*x*,*y*) is taken to be |*x*| + |*y*|.

vi is not referenced if *wi* = 0.0.

v

On exit, *v* contains the computed eigenvector, normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (*x*,*y*) is taken to be |*x*| + |*y*|.

info

`INTEGER`.

If *info* = 0, the execution is successful.

If *info* = 1, inverse iteration did not converge. For real flavors, *vr* is set to the last iterate, and so is *vi*, if *wi* ≠ 0.0. For complex flavors, *v* is set to the last iterate.

?laev2

Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.

Syntax

```
call slaev2( a, b, c, rt1, rt2, cs1, sn1 )
call dlaev2( a, b, c, rt1, rt2, cs1, sn1 )
call claev2( a, b, c, rt1, rt2, cs1, sn1 )
call zlaev2( a, b, c, rt1, rt2, cs1, sn1 )
```

Include Files

- `mkl.fi`

Description

The routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}$$ (for slaev2/dlaev2) or Hermitian matrix $$\begin{bmatrix} a & b \\ \text{conjg}(b) & c \end{bmatrix}$$

(for claev2/zlaev2).

On return, *rt1* is the eigenvalue of larger absolute value, *rt2* of smaller absolute value, and (*cs1*, *sn1*) is the unit right eigenvector for *rt1*, giving the decomposition

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for slaev2/dlaev2),

or

$$\begin{bmatrix} cs1 & \text{conjg}(sn1) \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ \text{conjg}(b) & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -\text{conjg}(sn1) \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for claev2/zlaev2).

Input Parameters

a, *b*, *c*

REAL for slaev2

DOUBLE PRECISION for dlaev2

COMPLEX for claev2

DOUBLE COMPLEX for zlaev2.

Elements of the input matrix.

Output Parameters

rt1, *rt2*

REAL for slaev2/claev2

DOUBLE PRECISION for dlaev2/zlaev2.

Eigenvalues of larger and smaller absolute value, respectively.

cs1

REAL for slaev2/claev2

DOUBLE PRECISION for dlaev2/zlaev2.

sn1

REAL for slaev2

DOUBLE PRECISION for dlaev2

COMPLEX for claev2

DOUBLE COMPLEX for zlaev2.

The vector (*cs1*, *sn1*) is the unit right eigenvector for rt1.

Application Notes

rt1 is accurate to a few ulps barring over/underflow. *rt2* may be inaccurate if there is massive cancellation in the determinant `a*c-b*b`; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute *rt2* accurately in all cases. *cs1* and *sn1* are accurate to a few ulps barring over/underflow. Overflow is possible only if *rt1* is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds *underflow_threshold* / `macheps`.

?laexc

Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.

Syntax

```
call slaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
call dlaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
```

Include Files

- `mkl.fi`

Description

The routine swaps adjacent diagonal blocks T_{11} and T_{22} of order 1 or 2 in an upper quasi-triangular matrix T by an orthogonal similarity transformation.

T must be in Schur canonical form, that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

wantq

`LOGICAL`.

If *wantq* = `.TRUE.`, accumulate the transformation in the matrix *Q*;

If *wantq* = `.FALSE.`, do not accumulate the transformation.

n

`INTEGER`. The order of the matrix *T* (*n*≥ `0`).

t, *q*

`REAL` for `slaexc`

`DOUBLE PRECISION` for `dlaexc`

Arrays:

t(*ldt*,`*`) contains on entry the upper quasi-triangular matrix *T*, in Schur canonical form.

The second dimension of *t* must be at least `max(1, n)`.

q(*ldq*,`*`) contains on entry, if *wantq* = `.TRUE.`, the orthogonal matrix *Q*. If *wantq* = `.FALSE.`, *q* is not referenced. The second dimension of *q* must be at least `max(1, n)`.

ldt

`INTEGER`. The leading dimension of *t*; at least `max(1, n)`.

ldq

`INTEGER`. The leading dimension of *q*;

If *wantq* = `.FALSE.`, then *ldq*≥ `1`.

If *wantq* = .TRUE., then *ldq* ≥ max(1,*n*).

j1 INTEGER. The index of the first row of the first block T_{11}.

n1 INTEGER. The order of the first block T_{11}

(*n1* = 0, 1, or 2).

n2 INTEGER. The order of the second block T_{22}

(*n2* = 0, 1, or 2).

work REAL for slaexc;

DOUBLE PRECISION for dlaexc.

Workspace array, DIMENSION (*n*).

Output Parameters

t On exit, the updated matrix *T*, again in Schur canonical form.

q On exit, if *wantq* = .TRUE., the updated matrix *Q*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, the transformed matrix *T* would be too far from Schur form; the blocks are not swapped and *T* and *Q* are unchanged.

?lag2

Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.

Syntax

```
call slag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
call dlag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
```

Include Files

- mkl.fi

Description

The routine computes the eigenvalues of a 2 x 2 generalized eigenvalue problem *A* - *w***B*, with scaling as necessary to avoid over-/underflow. The scaling factor, *s*, results in a modified eigenvalue equation

*s***A* - *w***B*,

where *s* is a non-negative scaling factor chosen so that *w*, *w***B*, and *s***A* do not overflow and, if possible, do not underflow, either.

Input Parameters

a, *b* REAL for slag2

DOUBLE PRECISION for dlag2

Arrays:

a(*lda*,2) contains, on entry, the 2 x 2 matrix *A*. It is assumed that its 1-norm is less than 1/*safmin*. Entries less than sqrt(*safmin*)*norm(*A*) are subject to being treated as zero.

b(*ldb*,2) contains, on entry, the 2 x 2 upper triangular matrix *B*. It is assumed that the one-norm of *B* is less than 1/*safmin*. The diagonals should be at least sqrt(*safmin*) times the largest element of *B* (in absolute value); if a diagonal is smaller than that, then +/- sqrt(*safmin*) will be used instead of that diagonal.

lda INTEGER. The leading dimension of *a*; *lda* ≥ 2.

ldb INTEGER. The leading dimension of *b*; *ldb* ≥ 2.

safmin REAL for slag2;

DOUBLE PRECISION for dlag2.

The smallest positive number such that 1/*safmin* does not overflow. (This should always be ?lamch('S') - it is an argument in order to avoid having to call ?lamch frequently.)

Output Parameters

scale1 REAL for slag2;

DOUBLE PRECISION for dlag2.

A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the first eigenvalue. If the eigenvalues are complex, then the eigenvalues are (*wr1* +/- *wii*)/*scale1* (which may lie outside the exponent range of the machine), *scale1*=*scale2*, and *scale1* will always be positive.

If the eigenvalues are real, then the first (real) eigenvalue is *wr1*/*scale1*, but this may overflow or underflow, and in fact, *scale1* may be zero or less than the underflow threshhold if the exact eigenvalue is sufficiently large.

scale2 REAL for slag2;

DOUBLE PRECISION for dlag2.

A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the second eigenvalue. If the eigenvalues are complex, then *scale2*=*scale1*. If the eigenvalues are real, then the second (real) eigenvalue is *wr2*/*scale2*, but this may overflow or underflow, and in fact, *scale2* may be zero or less than the underflow threshold if the exact eigenvalue is sufficiently large.

wr1 REAL for slag2;

DOUBLE PRECISION for dlag2.

If the eigenvalue is real, then *wr1* is *scale1* times the eigenvalue closest to the (2,2) element of *A**inv(*B*).

If the eigenvalue is complex, then *wr1=wr2* is *scale1* times the real part of the eigenvalues.

wr2

REAL for slag2;

DOUBLE PRECISION for dlag2.

If the eigenvalue is real, then *wr2* is *scale2* times the other eigenvalue. If the eigenvalue is complex, then *wr1=wr2* is *scale1* times the real part of the eigenvalues.

wi

REAL for slag2;

DOUBLE PRECISION for dlag2.

If the eigenvalue is real, then *wi* is zero. If the eigenvalue is complex, then *wi* is *scale1* times the imaginary part of the eigenvalues. *wi* will always be non-negative.

?lags2

Computes 2-by-2 orthogonal matrices U, V, and Q, and applies them to matrices A and B such that the rows of the transformed A and B are parallel.

Syntax

```
call slags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
call dlags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
call clags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
call zlags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
```

Include Files

- mkl.fi

Description

For real flavors, the routine computes 2-by-2 orthogonal matrices *U*, *V* and *Q*, such that if *upper* = .TRUE., then

$$U^T*A*Q = U^T*\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix}*Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

and

$$V^T*B*Q = V^T*\begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix}*Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

or if *upper* = .FALSE., then

$$U^T * A * Q = U^T * \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

and

$$V^T * B * Q = V^T * \begin{bmatrix} B_1 & 0 \\ B_2 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

The rows of the transformed *A* and *B* are parallel, where

$$U = \begin{bmatrix} csu & snu \\ -snu & csu \end{bmatrix}, \quad V = \begin{bmatrix} csv & snv \\ -snv & csv \end{bmatrix}, \quad Q = \begin{bmatrix} csq & snq \\ -snq & csq \end{bmatrix}$$

Here Z^T denotes the transpose of *Z*.

For complex flavors, the routine computes 2-by-2 unitary matrices *U*, *V* and *Q*, such that if `upper = .TRUE.`, then

$$U^H * A * Q = U^H * \begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix} * Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

and

$$V^H * B * Q = V^H * \begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

or if `upper = .FALSE.`, then

$$U^H * A * Q = U^H * \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

and

$$V^H * B * Q = V^H * \begin{bmatrix} B_1 & 0 \\ B_2 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

The rows of the transformed *A* and *B* are parallel, where

$$U = \begin{bmatrix} csu & snu \\ -snu^H & csu \end{bmatrix}, V = \begin{bmatrix} csv & snv \\ -snv^H & csv \end{bmatrix}, Q = \begin{bmatrix} csq & snq \\ -snq^H & csq \end{bmatrix}$$

Input Parameters

upper

LOGICAL.

If *upper* = .TRUE., the input matrices *A* and *B* are upper triangular; If *upper* = .FALSE., the input matrices *A* and *B* are lower triangular.

a1, *a3*

REAL for slags2 and clags2

DOUBLE PRECISION for dlags2 and zlags2

a2

REAL for slags2

DOUBLE PRECISION for dlags2

COMPLEX for clags2

COMPLEX*16 for zlags2

On entry, *a1, a2* and *a3* are elements of the input 2-by-2 upper (lower) triangular matrix *A*.

b1, *b3*

REAL for slags2 and clags2

DOUBLE PRECISION for dlags2 and zlags2

b2

REAL for slags2

DOUBLE PRECISION for dlags2

COMPLEX for clags2

COMPLEX*16 for zlags2

On entry, *b1, b2* and *b3* are elements of the input 2-by-2 upper (lower) triangular matrix *B*.

Output Parameters

csu

REAL for slags2 and clags2

DOUBLE PRECISION for dlags2 and zlags2

Element of the desired orthogonal matrix *U*.

snu

REAL for slags2

DOUBLE PRECISION for dlags2

Element of the desired orthogonal matrix *U*.

	COMPLEX for clags2
	COMPLEX*16 for zlags2
csv	REAL for slags2 and clags2
	DOUBLE PRECISION for dlags2 and zlags2
	Element of the desired orthogonal matrix *V*.
snv	REAL for slags2
	DOUBLE PRECISION for dlags2
	COMPLEX for clags2
	COMPLEX*16 for zlags2
	Element of the desired orthogonal matrix *V*.
csq	REAL for slags2 and clags2
	DOUBLE PRECISION for dlags2 and zlags2
	Element of the desired orthogonal matrix *Q*.
snq	REAL for slags2
	DOUBLE PRECISION for dlags2
	Element of the desired orthogonal matrix *Q*.
	COMPLEX for clags2
	COMPLEX*16 for zlags2

?lagtf

*Computes an LU factorization of a matrix T-λ*I, where T is a general tridiagonal matrix, and λ is a scalar, using partial pivoting with row interchanges.*

Syntax

```
call slagtf( n, a, lambda, b, c, tol, d, in, info )
call dlagtf( n, a, lambda, b, c, tol, d, in, info )
```

Include Files

- mkl.fi

Description

The routine factorizes the matrix (*T - lambda*I*), where *T* is an *n*-by-*n* tridiagonal matrix and *lambda* is a scalar, as

*T - lambda*I = P*L*U*,

where *P* is a permutation matrix, *L* is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and *U* is an upper triangular matrix with at most two non-zero super-diagonal elements per column. The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling. The parameter *lambda* is included in the routine so that ?lagtf may be used, in conjunction with ?lagts, to obtain eigenvectors of *T* by inverse iteration.

Input Parameters

n

INTEGER. The order of the matrix *T* (*n*≥ 0).

a, *b*, *c*

REAL for slagtf

DOUBLE PRECISION for dlagtf

Arrays, dimension *a*(*n*), *b*(*n*-1), *c*(*n*-1):

On entry, *a*(*) must contain the diagonal elements of the matrix *T*.

On entry, *b*(*) must contain the (*n*-1) super-diagonal elements of *T*.

On entry, *c*(*) must contain the (*n*-1) sub-diagonal elements of *T*.

tol

REAL for slagtf

DOUBLE PRECISION for dlagtf

On entry, a relative tolerance used to indicate whether or not the matrix (*T* - *lambda*I*) is nearly singular. *tol* should normally be chose as approximately the largest relative error in the elements of *T*. For example, if the elements of *T* are correct to about 4 significant figures, then *tol* should be set to about $5*10^{-4}$. If *tol* is supplied as less than eps, where eps is the relative machine precision, then the value eps is used in place of *tol*.

Output Parameters

a

On exit, *a* is overwritten by the *n* diagonal elements of the upper triangular matrix *U* of the factorization of *T*.

b

On exit, *b* is overwritten by the *n*-1 super-diagonal elements of the matrix *U* of the factorization of *T*.

c

On exit, *c* is overwritten by the *n*-1 sub-diagonal elements of the matrix *L* of the factorization of *T*.

d

REAL for slagtf

DOUBLE PRECISION for dlagtf

Array, dimension (*n*-2).

On exit, *d* is overwritten by the *n*-2 second super-diagonal elements of the matrix *U* of the factorization of *T*.

in

INTEGER.

Array, dimension (*n*).

On exit, *in* contains details of the permutation matrix *p*. If an interchange occurred at the k-th step of the elimination, then *in*(k) = 1, otherwise *in*(k) = 0. The element *in*(*n*) returns the smallest positive integer j such that

abs(*u*(j,j)) ≤ norm((*T* - *lambda*I*)(j))**tol*,

where norm(*A*(j)) denotes the sum of the absolute values of the j-th row of the matrix *A*.

If no such `j` exists then *in*(*n*) is returned as zero. If *in*(*n*) is returned as positive, then a diagonal element of *U* is small, indicating that (*T* – *lambda*I*) is singular or nearly singular.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*k*, the *k*-th parameter had an illegal value.

?lagtm

*Performs a matrix-matrix product of the form C = alpha*A*B+beta*C, where A is a tridiagonal matrix, B and C are rectangular matrices, and alpha and beta are scalars, which may be 0, 1, or -1.*

Syntax

```
call slagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call dlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call clagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call zlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
```

Include Files

- mkl.fi

Description

The routine performs a matrix-vector product of the form:

*B := alpha*A*X + beta*B*

where *A* is a tridiagonal matrix of order *n*, *B* and *X* are *n*-by-*nrhs* matrices, and *alpha* and *beta* are real scalars, each of which may be 0., 1., or -1.

Input Parameters

trans — CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', then *B := alpha*A*X + beta*B* (no transpose);

If *trans* = 'T', then *B := alpha*A*^{T}**X + beta*B* (transpose);

If *trans* = 'C', then *B := alpha*A*^{H}**X + beta*B* (conjugate transpose)

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

nrhs — INTEGER. The number of right-hand sides, i.e., the number of columns in *X* and *B* (*nrhs* ≥ 0).

alpha, *beta* — REAL for slagtm/clagtm

DOUBLE PRECISION for dlagtm/zlagtm

	Specify the scalars *alpha* and *beta* respectively. `alpha` must be 0., 1., or -1.; otherwise, it is assumed to be 0. `beta` must be 0., 1., or -1.; otherwise, it is assumed to be 1.

dl, *d*, *du*	`REAL` for `slagtm`
	`DOUBLE PRECISION` for `dlagtm`
	`COMPLEX` for `clagtm`
	`DOUBLE COMPLEX` for `zlagtm`.
	Arrays: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1).
	The array *dl* contains the (*n* - 1) sub-diagonal elements of *T*.
	The array *d* contains the *n* diagonal elements of *T*.
	The array *du* contains the (*n* - 1) super-diagonal elements of *T*.

x, *b*	`REAL` for `slagtm`
	`DOUBLE PRECISION` for `dlagtm`
	`COMPLEX` for `clagtm`
	`DOUBLE COMPLEX` for `zlagtm`.
	Arrays:
	`x(ldx,*)` contains the *n*-by-*nrhs* matrix *X*. The second dimension of *x* must be at least `max(1, nrhs)`.
	b(*ldb*,*) contains the *n*-by-*nrhs* matrix *B*. The second dimension of *b* must be at least `max(1, nrhs)`.

ldx	`INTEGER`. The leading dimension of the array *x*; *ldx*≥ `max(1, n)`.
ldb	`INTEGER`. The leading dimension of the array *b*; *ldb*≥ `max(1, n)`.

Output Parameters

b	Overwritten by the matrix expression `B := alpha*A*X + beta*B`

?lagts

*Solves the system of equations (T - lambda*I)*x = y or (T - lambda*I)^T*x = y,where T is a general tridiagonal matrix and lambda is a scalar, using the LU factorization computed by* `?lagtf`.

Syntax

```
call slagts( job, n, a, b, c, d, in, y, tol, info )
call dlagts( job, n, a, b, c, d, in, y, tol, info )
```

Include Files

- `mkl.fi`

Description

The routine may be used to solve for *x* one of the systems of equations:

`(T - lambda*I)*x = y` or `(T - lambda*I)`^{T}`*x = y`,

where *T* is an *n*-by-*n* tridiagonal matrix, following the factorization of (`T - lambda*I`) as

`T - lambda*I = P*L*U`,

computed by the routine `?lagtf`.

The choice of equation to be solved is controlled by the argument *job*, and in each case there is an option to perturb zero or very small diagonal elements of *U*, this option being intended for use in applications such as inverse iteration.

Input Parameters

job

`INTEGER`. Specifies the job to be performed by `?lagts` as follows:

= 1: The equations `(T - lambda*I)x = y` are to be solved, but diagonal elements of *U* are not to be perturbed.

= -1: The equations `(T - lambda*I)x = y` are to be solved and, if overflow would otherwise occur, the diagonal elements of *U* are to be perturbed. See argument *tol* below.

= 2: The equations `(T - lambda*I)`^{T}`x = y` are to be solved, but diagonal elements of *U* are not to be perturbed.

= -2: The equations `(T - lambda*I)`^{T}`x = y` are to be solved and, if overflow would otherwise occur, the diagonal elements of *U* are to be perturbed. See argument *tol* below.

n

`INTEGER`. The order of the matrix *T* (*n*≥ 0).

a, *b*, *c*, *d*

`REAL` for `slagts`

`DOUBLE PRECISION` for `dlagts`

Arrays, dimension *a*(*n*), *b*(*n*-1), *c*(*n*-1), *d*(*n*-2):

On entry, *a*`(*)` must contain the diagonal elements of *U* as returned from `?lagtf`.

On entry, *b*`(*)` must contain the first super-diagonal elements of *U* as returned from `?lagtf`.

On entry, *c*`(*)` must contain the sub-diagonal elements of *L* as returned from `?lagtf`.

On entry, *d*`(*)` must contain the second super-diagonal elements of *U* as returned from `?lagtf`.

in

`INTEGER`.

Array, dimension (*n*).

On entry, *in*`(*)` must contain details of the matrix *p* as returned from `?lagtf`.

y

`REAL` for `slagts`

`DOUBLE PRECISION` for `dlagts`

Array, dimension (*n*). On entry, the right hand side vector *y*.

tol — REAL for slagtf

DOUBLE PRECISION for dlagtf.

On entry, with *job* < 0, *tol* should be the minimum perturbation to be made to very small diagonal elements of *U*. *tol* should normally be chosen as about *eps**norm(*U*), where *eps* is the relative machine precision, but if *tol* is supplied as non-positive, then it is reset to *eps**max(abs(u(i,j))). If *job* > 0 then *tol* is not referenced.

Output Parameters

y — On exit, *y* is overwritten by the solution vector *x*.

tol — On exit, *tol* is changed as described in *Input Parameters* section above, only if *tol* is non-positive on entry. Otherwise *tol* is unchanged.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i* >0, overflow would occur when computing the *i*th element of the solution vector *x*. This can only occur when *job* is supplied as positive and either means that a diagonal element of *U* is very small, or that the elements of the right-hand side vector *y* are very large.

?lagv2

Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A,B) where B is upper triangular.

Syntax

```
call slagv2( a, lda, b, ldb, alphar, alphai, beta, csl, snl, csr, snr )
call dlagv2( a, lda, b, ldb, alphar, alphai, beta, csl, snl, csr, snr )
```

Include Files

- mkl.fi

Description

The routine computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (*A*,*B*) where *B* is upper triangular. The routine computes orthogonal (rotation) matrices given by *csl*, *snl* and *csr*, *snr* such that:

1) if the pencil (*A*,*B*) has two real eigenvalues (include 0/0 or 1/0 types), then

$$\begin{bmatrix} a_{11} & a_{11} \\ 0 & a_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & b_{11} \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

2) if the pencil (*A*,*B*) has a pair of complex conjugate eigenvalues, then

$$\begin{bmatrix} a_{11} & a_{11} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & 0 \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

where $b_{11} \geq b_{22} > 0$.

Input Parameters

a, *b*

`REAL` for `slagv2`

`DOUBLE PRECISION` for `dlagv2`

Arrays:

a(*lda*,`2`) contains the 2-by-2 matrix *A*;

b(*ldb*,`2`) contains the upper triangular 2-by-2 matrix *B*.

lda

`INTEGER`. The leading dimension of the array *a*;

lda≥ `2`.

ldb

`INTEGER`. The leading dimension of the array *b*;

ldb≥ `2`.

Output Parameters

a

On exit, *a* is overwritten by the "*A*-part" of the generalized Schur form.

b

On exit, *b* is overwritten by the "*B*-part" of the generalized Schur form.

alphar, *alphai*, *beta*

`REAL` for `slagv2`

`DOUBLE PRECISION` for `dlagv2`.

Arrays, dimension (2) each.

(*alphar*(k) + *i***alphai*(k))/*beta*(k) are the eigenvalues of the pencil (*A*,*B*), k=1,2 and *i* = sqrt(-1).

Note that *beta*(k) may be zero.

csl, *snl*

`REAL` for `slagv2`

`DOUBLE PRECISION` for `dlagv2`

The cosine and sine of the left rotation matrix, respectively.

csr, *snr*

REAL for slagv2

DOUBLE PRECISION for dlagv2

The cosine and sine of the right rotation matrix, respectively.

?lahqr

Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/ single-shift QR algorithm.

Syntax

```
call slahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, info )
call dlahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, info )
call clahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
call zlahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
```

Include Files

- mkl.fi

Description

The routine is an auxiliary routine called by ?hseqr to update the eigenvalues and Schur decomposition already computed by ?hseqr, by dealing with the Hessenberg submatrix in rows and columns *ilo* to *ihi*.

Input Parameters

wantt

LOGICAL.

If *wantt* = .TRUE., the full Schur form *T* is required;

If *wantt* = .FALSE., eigenvalues only are required.

wantz

LOGICAL.

If *wantz* = .TRUE., the matrix of Schur vectors *Z* is required;

If *wantz* = .FALSE., Schur vectors are not required.

n

INTEGER. The order of the matrix *H* ($n \geq 0$).

ilo, *ihi*

INTEGER.

It is assumed that *h* is already upper quasi-triangular in rows and columns *ihi*+1:*n*, and that *h*(*ilo*,*ilo*-1) = 0 (unless *ilo* = 1). The routine ?lahqr works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *h* if *wantt* = .TRUE..

Constraints:

$1 \leq ilo \leq \max(1, ihi)$; $ihi \leq n$.

h, *z*

REAL for slahqr

DOUBLE PRECISION for dlahqr

COMPLEX for clahqr

DOUBLE COMPLEX for zlahqr.

Arrays:

h(*ldh*,*) contains the upper Hessenberg matrix *h*.

The second dimension of *h* must be at least max(1, *n*).

z(*ldz*,*)

If *wantz* = .TRUE., then, on entry, *z* must contain the current matrix *z* of transformations accumulated by ?hseqr. The second dimension of *z* must be at least max(1, *n*)

If *wantz* = .FALSE., then *z* is not referenced..

ldh — INTEGER. The leading dimension of *h*; at least max(1, *n*).

ldz — INTEGER. The leading dimension of *z*; at least max(1, *n*).

iloz, *ihiz* — INTEGER. Specify the rows of *z* to which transformations must be applied if *wantz* = .TRUE..

1 ≤ *iloz* ≤ *ilo*; *ihi* ≤ *ihiz* ≤ *n*.

Output Parameters

h

On exit, if *info*= 0 and *wantt* = .TRUE., then,

- for slahqr/dlahqr, *h* is upper quasi-triangular in rows and columns *ilo*:*ihi* with any 2-by-2 diagonal blocks in standard form.
- for clahqr/zlahqr, *h* is upper triangular in rows and columns *ilo*:*ihi*.

If *info*= 0 and *wantt* = .FALSE., the contents of *h* are unspecified on exit. If *info* is positive, see description of *info* for the output state of *h*.

wr, *wi*

REAL for slahqr

DOUBLE PRECISION for dlahqr

Arrays, DIMENSION at least max(1, *n*) each. Used with real flavors only. The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the i-th and (i+1)-th, with *wi*(i)> 0 and *wi*(i+1) < 0.

If *wantt* = .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *wr*(i) = *h*(i,i), and, if *h*(i:i+1, i:i+1) is a 2-by-2 diagonal block, *wi*(i) = sqrt(*h*(i +1,i)*h(i,i+1)) and *wi*(i+1) = -*wi*(i).

w

COMPLEX for clahqr

DOUBLE COMPLEX for zlahqr.

Array, DIMENSION at least max(1, *n*). Used with complex flavors only. The computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *w*.

If *wantt* = .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with w(i) = h(i,i).

z — If *wantz* = .TRUE., then, on exit *z* has been updated; transformations are applied only to the submatrix *z*(*iloz*:*ihiz*, *ilo*:*ihi*).

info — INTEGER.

If *info* = 0, the execution is successful.

With *info* > 0,

- if *info* = *i*, ?lahqr failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30 iterations per eigenvalue; elements i+1:*ihi* of *wr* and *wi* (for slahqr/dlahqr) or *w* (for clahqr/zlahqr) contain those eigenvalues which have been successfully computed.
- if *wantt* is .FALSE., then on exit the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.
- if *wantt* is .TRUE., then on exit
(initial value of *h*)*u = u*(final value of *h*), (*)
where u is an orthognal matrix. The final value of *h* is upper Hessenberg and triangular in rows and columns *info*+1 through *ihi*.
- if *wantz* is .TRUE., then on exit
(final value of *z*) = (initial value of *z*)* u,
where u is an orthognal matrix in (*) regardless of the value of *wantt*.

?lahrd

Reduces the first nb columns of a general rectangular matrix A so that elements below the k-th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A (deprecated).

Syntax

```
call slahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Include Files

- mkl.fi

Description

This routine is deprecated; use lahr2.

The routine reduces the first *nb* columns of a real/complex general *n*-by-(*n*-*k*+1) matrix *A* so that elements below the *k*-th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation Q^T*A*Q for real flavors, or Q^H*A*Q for complex flavors. The routine returns the matrices *V* and *T* which determine *Q* as a block reflector $I - V*T*V^T$ (for real flavors) or $I - V*T*V^H$ (for complex flavors), and also the matrix $Y = A*V*T$.

The matrix *Q* is represented as products of *nb* elementary reflectors:

$Q = H(1)*H(2)*... *H(nb)$

Each $H(i)$ has the form

$H(i) = I - tau*v*v^T$ for real flavors, or

$H(i) = I - tau*v*v^H$ for complex flavors, or

where *tau* is a real/complex scalar, and *v* is a real/complex vector.

Input Parameters

n — INTEGER. The order of the matrix *A* ($n \geq 0$).

k — INTEGER. The offset for the reduction. Elements below the *k*-th subdiagonal in the first *nb* columns are reduced to zero.

nb — INTEGER. The number of columns to be reduced.

a — REAL for slahrd

DOUBLE PRECISION for dlahrd

COMPLEX for clahrd

DOUBLE COMPLEX for zlahrd.

Array *a*(*lda*, *n*-*k*+1) contains the *n*-by-(*n*-*k*+1) general matrix *A* to be reduced.

lda — INTEGER. The leading dimension of *a*; at least max(1, *n*).

ldt — INTEGER. The leading dimension of the output array *t*; must be at least max(1, *nb*).

ldy — INTEGER. The leading dimension of the output array *y*; must be at least max(1, *n*).

Output Parameters

a — On exit, the elements on and above the *k*-th subdiagonal in the first *nb* columns are overwritten with the corresponding elements of the reduced matrix; the elements below the *k*-th subdiagonal, with the array *tau*, represent the matrix *Q* as a product of elementary reflectors. The other columns of *a* are unchanged. See *Application Notes* below.

tau — REAL for slahrd

DOUBLE PRECISION for dlahrd

COMPLEX for clahrd

DOUBLE COMPLEX for zlahrd.

Array, DIMENSION (*nb*).

Contains scalar factors of the elementary reflectors.

t, *y* — REAL for slahrd

DOUBLE PRECISION for dlahrd

COMPLEX for clahrd

DOUBLE COMPLEX for zlahrd.

Arrays, dimension *t*(*ldt*, *nb*), *y*(*ldy*, *nb*).

The array *t* contains upper triangular matrix *T*.

The array *y* contains the *n*-by-*nb* matrix *Y* .

Application Notes

For the elementary reflector *H*(i),

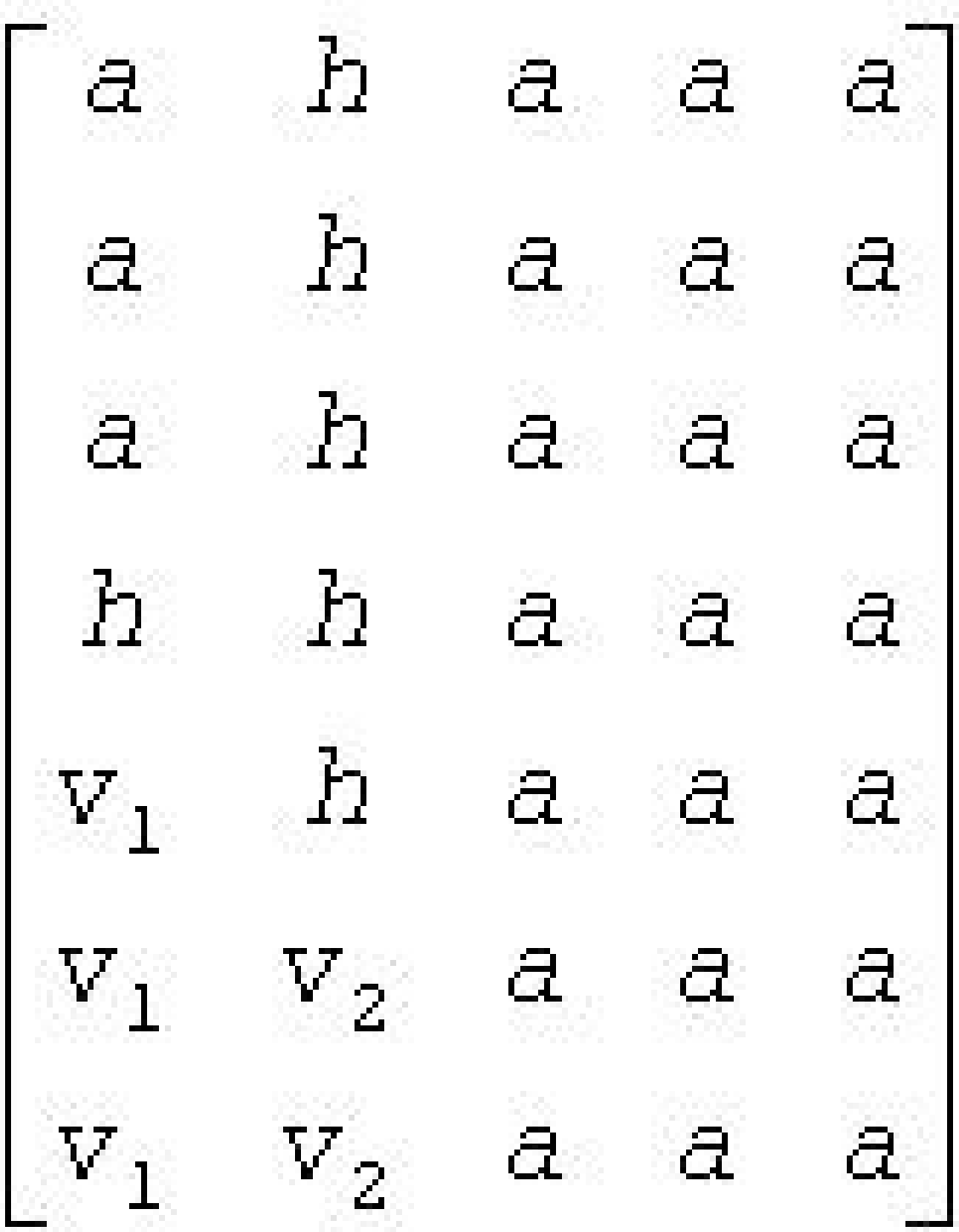
`v(1:i+k-1) = 0`, `v(i+k) = 1`; `v(i+k+1:n)` is stored on exit in *a*(i+*k*+1:*n*, i) and *tau* is stored in *tau*(i).

The elements of the vectors *v* together form the (*n*-*k*+1)-by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form:

$A := (I - V*T*V^T) * (A - Y*V^T)$ for real flavors, or

$A := (I - V*T*V^H) * (A - Y*V^H)$ for complex flavors.

The contents of *A* on exit are illustrated by the following example with $n = 7$, $k = 3$ and $nb = 2$:



where *a* denotes an element of the original matrix *A*, *h* denotes a modified element of the upper Hessenberg matrix *H*, and v_i denotes an element of the vector defining *H*(i).

See Also

?lahr2

?lahr2

Reduces the specified number of first columns of a general rectangular matrix A so that elements below the specified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A.

Syntax

```
call slahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Include Files

- mkl.fi

Description

The routine reduces the first *nb* columns of a real/complex general *n*-by-(*n*-*k*+1) matrix *A* so that elements below the *k*-th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation Q^T*A*Q for real flavors, or Q^H*A*Q for complex flavors. The routine returns the matrices *V* and *T* which determine *Q* as a block reflector $I - V*T*V^T$ (for real flavors) or $I - V*T*V^H$ (for real flavors), and also the matrix $Y = A*V*T$.

The matrix *Q* is represented as products of *nb* elementary reflectors:

$Q = H(1)*H(2)*\ldots *H(nb)$

Each $H(i)$ has the form

$H(i) = I - tau*v*v^T$ for real flavors, or

$H(i) = I - tau*v*v^H$ for complex flavors

where *tau* is a real/complex scalar, and *v* is a real/complex vector.

This is an auxiliary routine called by ?gehrd.

Input Parameters

n	INTEGER. The order of the matrix *A* ($n \geq 0$).
k	INTEGER. The offset for the reduction. Elements below the *k*-th subdiagonal in the first *nb* columns are reduced to zero ($k < n$).
nb	INTEGER. The number of columns to be reduced.
a	REAL for slahr2 DOUBLE PRECISION for dlahr2 COMPLEX for clahr2 DOUBLE COMPLEX for zlahr2. Array, DIMENSION (*lda*, *n*-*k*+1) contains the *n*-by-(*n*-*k*+1) general matrix *A* to be reduced.

lda	INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1, *n*).
ldt	INTEGER. The leading dimension of the output array *t*; *ldt* ≥ *nb*.
ldy	INTEGER. The leading dimension of the output array *y*; *ldy* ≥ *n*.

Output Parameters

a	On exit, the elements on and above the *k*-th subdiagonal in the first *nb* columns are overwritten with the corresponding elements of the reduced matrix; the elements below the *k*-th subdiagonal, with the array *tau*, represent the matrix *Q* as a product of elementary reflectors. The other columns of *a* are unchanged. See *Application Notes* below.
tau	REAL for slahr2 DOUBLE PRECISION for dlahr2 COMPLEX for clahr2 DOUBLE COMPLEX for zlahr2. Array, DIMENSION (*nb*). Contains scalar factors of the elementary reflectors.
t, *y*	REAL for slahr2 DOUBLE PRECISION for dlahr2 COMPLEX for clahr2 DOUBLE COMPLEX for zlahr2. Arrays, dimension *t*(*ldt*, *nb*), *y*(*ldy*, *nb*). The array *t* contains upper triangular matrix *T*. The array *y* contains the *n*-by-*nb* matrix *Y* .

Application Notes

For the elementary reflector *H*(i),

v(1:i+*k*-1) = 0, *v*(i+*k*) = 1; *v*(i+*k*+1:*n*) is stored on exit in *a*(i+*k*+1:*n*, i) and *tau* is stored in *tau*(i).

The elements of the vectors *v* together form the (*n*-*k*+1)-by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form:

$A := (I - V*T*V^T) * (A - Y*V^T)$ for real flavors, or

$A := (I - V*T*V^H) * (A - Y*V^H)$ for complex flavors.

The contents of *A* on exit are illustrated by the following example with *n* = 7, *k* = 3 and *nb* = 2:

where *a* denotes an element of the original matrix *A*, *h* denotes a modified element of the upper Hessenberg matrix *H*, and v_i denotes an element of the vector defining `H(i)`.

?laic1

Applies one step of incremental condition estimation.

Syntax

```
call slaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call dlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call claic1( job, j, x, sest, w, gamma, sestpr, s, c )
call zlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
```

Include Files

- mkl.fi

Description

The routine ?laic1 applies one step of incremental condition estimation in its simplest version.

Let *x*, $||x||_2 = 1$ (where $||a||_2$ denotes the 2-norm of *a*), be an approximate singular vector of an *j*-by-*j* lower triangular matrix *L*, such that

$||L*x||_2 = sest$

Then ?laic1 computes *sestpr*, *s*, *c* such that the vector

$$xhat = \begin{bmatrix} s*x \\ c \end{bmatrix}$$

is an approximate singular vector of

$$Lhat = \begin{bmatrix} L & 0 \\ w^H & gamma \end{bmatrix}$$

(for complex flavors), or

$$Lhat = \begin{bmatrix} L & 0 \\ w^T & gamma \end{bmatrix}$$

(for real flavors), in the sense that

$||Lhat*xhat||_2 = sestpr$.

Depending on *job*, an estimate for the largest or smallest singular value is computed.

For real flavors, $[sc]^T$ and $sestpr^2$ is an eigenpair of the system

$$\mathrm{diag}(sest*sest, 0) + [alpha\ gamma] * \begin{bmatrix} alpha \\ gamma \end{bmatrix}$$

where $alpha = x^T*w$.

For complex flavors, $[sc]^H$ and $sestpr^2$ is an eigenpair of the system

$$\mathrm{diag}(sest*sest, 0) + [alpha\ gamma] * \begin{bmatrix} \mathrm{conjg}(alpha) \\ \mathrm{conjg}(gamma) \end{bmatrix}$$

where *alpha* = $x^{H} * w$.

Input Parameters

job	INTEGER. If *job* =1, an estimate for the largest singular value is computed; If *job* =2, an estimate for the smallest singular value is computed;
j	INTEGER. Length of *x* and *w*.
x, *w*	REAL for slaic1 DOUBLE PRECISION for dlaic1 COMPLEX for claic1 DOUBLE COMPLEX for zlaic1. Arrays, dimension (*j*) each. Contain vectors *x* and *w*, respectively.
sest	REAL for slaic1/claic1; DOUBLE PRECISION for dlaic1/zlaic1. Estimated singular value of *j*-by-*j* matrix *L*.
gamma	REAL for slaic1 DOUBLE PRECISION for dlaic1 COMPLEX for claic1 DOUBLE COMPLEX for zlaic1. The diagonal element *gamma*.

Output Parameters

sestpr	REAL for slaic1/claic1; DOUBLE PRECISION for dlaic1/zlaic1. Estimated singular value of (*j*+1)-by-(*j*+1) matrix *Lhat*.
s, *c*	REAL for slaic1 DOUBLE PRECISION for dlaic1 COMPLEX for claic1 DOUBLE COMPLEX for zlaic1. Sine and cosine needed in forming *xhat*.

?lakf2

Forms a matrix containing Kronecker products between the given matrices.

Syntax

```
call slakf2( m, n, a, lda, b, d, e, z, ldz )
call dlakf2( m, n, a, lda, b, d, e, z, ldz )
```

```
call clakf2( m, n, a, lda, b, d, e, z, ldz )
call zlakf2( m, n, a, lda, b, d, e, z, ldz )
```

Include Files

- mkl.fi

Description

The routine ?lakf2 forms the 2*m*n by 2*m*n matrix *Z*.

$$Z = \begin{bmatrix} \text{kron}(In, A) & -\text{kron}(B^T, Im) \\ \text{kron}(In, D) & -\text{kron}(E^T, Im) \end{bmatrix}$$

,

where *In* is the identity matrix of size *n* and X^T is the transpose of *X*. kron(*X*, *Y*) is the Kronecker product between the matrices *X* and *Y*.

Input Parameters

m	INTEGER. Size of matrix, $m \geq 1$
n	INTEGER. Size of matrix, $n \geq 1$
a	REAL for slakf2, DOUBLE PRECISION for dlakf2, COMPLEX for clakf2, DOUBLE COMPLEX for zlakf2, Array, size *lda*-by-*n*. The matrix *A* in the output matrix *Z*.
lda	INTEGER. The leading dimension of *a*, *b*, *d*, and *e*. $lda \geq m+n$.
b	REAL for slakf2, DOUBLE PRECISION for dlakf2, COMPLEX for clakf2, DOUBLE COMPLEX for zlakf2, Array, size *lda* by *n*. Matrix used in forming the output matrix *Z*.
d	REAL for slakf2, DOUBLE PRECISION for dlakf2, COMPLEX for clakf2, DOUBLE COMPLEX for zlakf2, Array, size *lda* by *m*. Matrix used in forming the output matrix *Z*.
e	REAL for slakf2, DOUBLE PRECISION for dlakf2, COMPLEX for clakf2, DOUBLE COMPLEX for zlakf2,

	Array, size *lda* by *n*. Matrix used in forming the output matrix *Z*.
ldz	INTEGER. The leading dimension of *Z*. *ldz*≥ 2* *m*n*.

Output Parameters

z	REAL for slakf2, DOUBLE PRECISION for dlakf2, COMPLEX for clakf2, DOUBLE COMPLEX for zlakf2, Array, size *ldz*-by-2**m***n*. The resultant Kronecker *m***n*2 -by-*m***n*2 matrix.

?laln2

Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.

Syntax

```
call slaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx, scale, xnorm, info )
call dlaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx, scale, xnorm, info )
```

Include Files

- mkl.fi

Description

The routine solves a system of the form

$(ca*A - w*D)*X = s*B$, or $(ca*A^T - w*D)*X = s*B$

with possible scaling (*s*) and perturbation of *A*.

A is an *na*-by-*na* real matrix, *ca* is a real scalar, *D* is an *na*-by-*na* real diagonal matrix, *w* is a real or complex value, and *X* and *B* are *na*-by-1 matrices: real if *w* is real, complex if *w* is complex. The parameter *na* may be 1 or 2.

If *w* is complex, *X* and *B* are represented as *na*-by-2 matrices, the first column of each being the real part and the second being the imaginary part.

The routine computes the scaling factor s (≤ 1) so chosen that *X* can be computed without overflow. *X* is further scaled if necessary to assure that norm(*ca*A - w*D*)*norm(*X*) is less than overflow.

If both singular values of (*ca*A - w*D*) are less than *smin*, *smin***I* (where *I* stands for identity) will be used instead of (*ca***A* - *w***D*). If only one singular value is less than *smin*, one element of (*ca***A* - *w***D*) will be perturbed enough to make the smallest singular value roughly *smin*.

If both singular values are at least *smin*, (*ca*A - w*D*) will not be perturbed. In any case, the perturbation will be at most some small multiple of max(*smin*, *ulp**norm(*ca*A - w*D*)).

The singular values are computed by infinity-norm approximations, and thus will only be correct to a factor of 2 or so.

NOTE
All input quantities are assumed to be smaller than overflow by a reasonable factor (see *bignum*).

Input Parameters

trans LOGICAL.

If *trans* = .TRUE., *A*- transpose will be used.

If *trans* = .FALSE., *A* will be used (not transposed.)

na INTEGER. The size of the matrix *A*, possible values 1 or 2.

nw INTEGER. This parameter must be 1 if *w* is real, and 2 if *w* is complex. Possible values 1 or 2.

smin REAL for slaln2

DOUBLE PRECISION for dlaln2.

The desired lower bound on the singular values of *A*.

This should be a safe distance away from underflow or overflow, for example, between (*underflow/machine_precision*) and (*machine_precision* * *overflow*). (See *bignum* and *ulp*).

ca REAL for slaln2

DOUBLE PRECISION for dlaln2.

The coefficient by which *A* is multiplied.

a REAL for slaln2

DOUBLE PRECISION for dlaln2.

Array, DIMENSION (*lda*,*na*).

The *na*-by-*na* matrix *A*.

lda INTEGER. The leading dimension of *a*. Must be at least *na*.

d1, *d2* REAL for slaln2

DOUBLE PRECISION for dlaln2.

The (1,1) and (2,2) elements in the diagonal matrix *D*, respectively. *d2* is not used if *nw* = 1.

b REAL for slaln2

DOUBLE PRECISION for dlaln2.

Array, DIMENSION (*ldb*,*nw*). The *na*-by-*nw* matrix *B* (right-hand side). If *nw* =2 (*w* is complex), column 1 contains the real part of *B* and column 2 contains the imaginary part.

ldb INTEGER. The leading dimension of *b*. Must be at least *na*.

wr, *wi* REAL for slaln2

DOUBLE PRECISION for dlaln2.

The real and imaginary part of the scalar *w*, respectively.

wi is not used if *nw* = 1.

ldx INTEGER. The leading dimension of the output array *x*. Must be at least *na*.

Output Parameters

x REAL for slaln2

DOUBLE PRECISION for dlaln2.

Array, DIMENSION (*ldx*,*nw*). The *na*-by-*nw* matrix *X* (unknowns), as computed by the routine. If *nw* = 2 (*w* is complex), on exit, column 1 will contain the real part of *X* and column 2 will contain the imaginary part.

scale REAL for slaln2

DOUBLE PRECISION for dlaln2.

The scale factor that *B* must be multiplied by to insure that overflow does not occur when computing *X*. Thus (*ca*A - w*D*) *X* will be *scale***B*, not *B* (ignoring perturbations of *A*.) It will be at most 1.

xnorm REAL for slaln2

DOUBLE PRECISION for dlaln2.

The infinity-norm of *X*, when *X* is regarded as an *na*-by-*nw* real matrix.

info INTEGER.

An error flag. It will be zero if no error occurs, a negative number if an argument is in error, or a positive number if (*ca*A - w*D*) had to be perturbed. The possible values are:

If *info* = 0: no error occurred, and (*ca*A - w*D*) did not have to be perturbed.

If *info* = 1: (*ca*A - w*D*) had to be perturbed to make its smallest (or only) singular value greater than *smin*.

NOTE
For higher speed, this routine does not check the inputs for errors.

?lals0

Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd.

Syntax

call slals0(*icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info*)

call dlals0(*icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info*)

call clals0(*icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info*)

```
call zlals0( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol, ldgcol,
givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info )
```

Include Files

- mkl.fi

Description

The routine applies back the multiplying factors of either the left or right singular vector matrix of a diagonal matrix appended by a row to the right hand side matrix *B* in solving the least squares problem using the divide-and-conquer SVD approach.

For the left singular vector matrix, three types of orthogonal matrices are involved:

(1L) Givens rotations: the number of such rotations is *givptr*;the pairs of columns/rows they were applied to are stored in *givcol*;and the *c*- and *s*-values of these rotations are stored in *givnum*.

(2L) Permutation. The (*nl*+1)-st row of *B* is to be moved to the first row, and for j=2:*n*, *perm*(*j*)-th row of *B* is to be moved to the *j*-th row.

(3L) The left singular vector matrix of the remaining matrix.

For the right singular vector matrix, four types of orthogonal matrices are involved:

(1R) The right singular vector matrix of the remaining matrix.

(2R) If *sqre* = 1, one extra Givens rotation to generate the right null space.

(3R) The inverse transformation of (2L).

(4R) The inverse transformation of (1L).

Input Parameters

icompq

INTEGER. Specifies whether singular vectors are to be computed in factored form:

If *icompq* = 0: Left singular vector matrix.

If *icompq* = 1: Right singular vector matrix.

nl

INTEGER. The row dimension of the upper block.

nl≥ 1.

nr

INTEGER. The row dimension of the lower block.

nr≥ 1.

sqre

INTEGER.

If *sqre* = 0: the lower block is an *nr*-by-*nr* square matrix.

If *sqre* = 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has row dimension *n* = *nl* + *nr* + 1, and column dimension *m* = *n* + *sqre*.

nrhs

INTEGER. The number of columns of *B* and *bx*.

Must be at least 1.

b

REAL for slals0

DOUBLE PRECISION for dlals0

COMPLEX for clals0

DOUBLE COMPLEX for zlals0.

Array, DIMENSION (*ldb*, *nrhs*).

Contains the right hand sides of the least squares problem in rows 1 through *m*.

ldb INTEGER. The leading dimension of *b*.

Must be at least max(1,max(*m*, *n*)).

bx REAL for slals0

DOUBLE PRECISION for dlals0

COMPLEX for clals0

DOUBLE COMPLEX for zlals0.

Workspace array, DIMENSION (*ldbx*, *nrhs*).

ldbx INTEGER. The leading dimension of *bx*.

perm INTEGER. Array, DIMENSION (*n*).

The permutations (from deflation and sorting) applied to the two blocks.

givptr INTEGER. The number of Givens rotations which took place in this subproblem.

givcol INTEGER. Array, DIMENSION (*ldgcol*, 2). Each pair of numbers indicates a pair of rows/columns involved in a Givens rotation.

ldgcol INTEGER. The leading dimension of *givcol*, must be at least *n*.

givnum REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Array, DIMENSION (*ldgnum*, 2). Each number indicates the c or *s* value used in the corresponding Givens rotation.

ldgnum INTEGER. The leading dimension of arrays *difr*, *poles* and *givnum*, must be at least *k*.

poles REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Array, DIMENSION (*ldgnum*, 2). On entry, *poles*(1:*k*, 1) contains the new singular values obtained from solving the secular equation, and *poles*(1:*k*, 2) is an array containing the poles in the secular equation.

difl REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Array, DIMENSION (*k*). On entry, *difl*(*i*) is the distance between *i*-th updated (undeflated) singular value and the *i*-th (undeflated) old singular value.

difr REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Array, DIMENSION (*ldgnum*, 2). On entry, *difr*(*i*, 1) contains the distances between *i*-th updated (undeflated) singular value and the *i+1*-th (undeflated) old singular value. And *difr*(*i*, 2) is the normalizing factor for the *i*-th right singular vector.

z

REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Array, DIMENSION (*k*). Contains the components of the deflation-adjusted updating row vector.

K

INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \leq k \leq n$.

c

REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Contains garbage if *sqre* =0 and the *c* value of a Givens rotation related to the right null space if *sqre* = 1.

s

REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Contains garbage if *sqre* =0 and the *s* value of a Givens rotation related to the right null space if *sqre* = 1.

work

REAL for slals0

DOUBLE PRECISION for dlals0

Workspace array, DIMENSION (*k*). Used with real flavors only.

rwork

REAL for clals0

DOUBLE PRECISION for zlals0

Workspace array, DIMENSION (*k**(1+*nrhs*) + 2**nrhs*). Used with complex flavors only.

Output Parameters

b

On exit, contains the solution *X* in rows 1 through *n*.

info

INTEGER.

If *info* = 0: successful exit.

If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lalsa

Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.

Syntax

```
call slalsa( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

```
call dlalsa( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )

call clalsa( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork, info )

call zlalsa( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork, info )
```

Include Files

- mkl.fi

Description

The routine is an intermediate step in solving the least squares problem by computing the SVD of the coefficient matrix in compact form. The singular vectors are computed as products of simple orthogonal matrices.

If *icompq* = 0, ?lalsa applies the inverse of the left singular vector matrix of an upper bidiagonal matrix to the right hand side; and if *icompq* = 1, the routine applies the right singular vector matrix to the right hand side. The singular vector matrices were generated in the compact form by ?lalsa.

Input Parameters

icompq
: INTEGER. Specifies whether the left or the right singular vector matrix is involved. If *icompq* = 0: left singular vector matrix is used

  If *icompq* = 1: right singular vector matrix is used.

smlsiz
: INTEGER. The maximum size of the subproblems at the bottom of the computation tree.

n
: INTEGER. The row and column dimensions of the upper bidiagonal matrix.

nrhs
: INTEGER. The number of columns of *b* and *bx*. Must be at least 1.

b
: REAL for slalsa

  DOUBLE PRECISION for dlalsa

  COMPLEX for clalsa

  DOUBLE COMPLEX for zlalsa

  Array, DIMENSION (*ldb*, *nrhs*). Contains the right hand sides of the least squares problem in rows 1 through *m*.

ldb
: INTEGER. The leading dimension of *b* in the calling subprogram. Must be at least max(1,max(*m*, *n*)).

ldbx
: INTEGER. The leading dimension of the output array *bx*.

u
: REAL for slalsa/clalsa

  DOUBLE PRECISION for dlalsa/zlalsa

  Array, DIMENSION (*ldu*, *smlsiz*). On entry, *u* contains the left singular vector matrices of all subproblems at the bottom level.

ldu INTEGER, *ldu*≥*n*. The leading dimension of arrays *u*, *vt*, *difl*, *difr*, *poles*, *givnum*, and *z*.

vt REAL for slalsa/clalsa

DOUBLE PRECISION for dlalsa/zlalsa

Array, DIMENSION(*ldu*, *smlsiz* +1). On entry, vt^T (for real flavors) or vt^H (for complex flavors) contains the right singular vector matrices of all subproblems at the bottom level.

k INTEGER array, DIMENSION (*n*).

difl REAL for slalsa/clalsa

DOUBLE PRECISION for dlalsa/zlalsa

Array, DIMENSION (*ldu*, *nlvl*), where $nlvl = \text{int}(\log_2(n/(smlsiz+1))) + 1$.

difr REAL for slalsa/clalsa

DOUBLE PRECISION for dlalsa/zlalsa

Array, DIMENSION(*ldu*, 2**nlvl*). On entry, *difl*(*, *i*) and *difr*(*, 2*i* -1) record distances between singular values on the *i*-th level and singular values on the (*i* -1)-th level, and *difr*(*, 2*i*) record the normalizing factors of the right singular vectors matrices of subproblems on *i*-th level.

z REAL for slalsa/clalsa

DOUBLE PRECISION for dlalsa/zlalsa

Array, DIMENSION (*ldu*, *nlvl* . On entry, *z*(1, *i*) contains the components of the deflation- adjusted updating the row vector for subproblems on the *i*-th level.

poles REAL for slalsa/clalsa

DOUBLE PRECISION for dlalsa/zlalsa

Array, DIMENSION (*ldu*, 2**nlvl*).

On entry, *poles*(*, 2*i*-1: 2*i*) contains the new and old singular values involved in the secular equations on the *i*-th level.

givptr INTEGER. Array, DIMENSION (*n*).

On entry, *givptr*(*i*) records the number of Givens rotations performed on the *i*-th problem on the computation tree.

givcol INTEGER. Array, DIMENSION (*ldgcol*, 2**nlvl*). On entry, for each *i*, *givcol*(*, 2*i*-1: 2*i*) records the locations of Givens rotations performed on the *i*-th level on the computation tree.

ldgcol INTEGER, *ldgcol*≥*n*. The leading dimension of arrays *givcol* and *perm*.

perm INTEGER. Array, DIMENSION (*ldgcol*, *nlvl*). On entry, *perm*(*, *i*) records permutations done on the *i*-th level of the computation tree.

givnum REAL for slalsa/clalsa

DOUBLE PRECISION for dlalsa/zlalsa

Array, DIMENSION (*ldu*, 2**nlvl*). On entry, *givnum*(*, 2*i*-1 : 2*i*) records the *c* and *s* values of Givens rotations performed on the *i*-th level on the computation tree.

c

REAL for slalsa/clalsa

DOUBLE PRECISION for dlalsa/zlalsa

Array, DIMENSION (*n*). On entry, if the *i*-th subproblem is not square, *c*(*i*) contains the *c* value of a Givens rotation related to the right null space of the *i*-th subproblem.

s

REAL for slalsa/clalsa

DOUBLE PRECISION for dlalsa/zlalsa

Array, DIMENSION (*n*). On entry, if the *i*-th subproblem is not square, *s*(*i*) contains the *s*-value of a Givens rotation related to the right null space of the *i*-th subproblem.

work

REAL for slalsa

DOUBLE PRECISION for dlalsa

Workspace array, DIMENSION at least (*n*). Used with real flavors only.

rwork

REAL for clalsa

DOUBLE PRECISION for zlalsa

Workspace array, DIMENSION at least max(*n*, (*smlsz*+1)**nrhs**3). Used with complex flavors only.

iwork

INTEGER.

Workspace array, DIMENSION at least (3*n*).

Output Parameters

b

On exit, contains the solution *X* in rows 1 through *n*.

bx

REAL for slalsa

DOUBLE PRECISION for dlalsa

COMPLEX for clalsa

DOUBLE COMPLEX for zlalsa

Array, DIMENSION (*ldbx*, *nrhs*). On exit, the result of applying the left or right singular vector matrix to *b*.

info

INTEGER. If *info* = 0: successful exit

If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lalsd

Uses the singular value decomposition of A to solve the least squares problem.

Syntax

```
call slalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork, info )
```

```
call dlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork, info )
call clalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork, iwork, info )
call zlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork, iwork, info )
```

Include Files

- mkl.fi

Description

The routine uses the singular value decomposition of *A* to solve the least squares problem of finding *X* to minimize the Euclidean norm of each column of *A***X*-*B*, where *A* is *n*-by-*n* upper bidiagonal, and *X* and *B* are *n*-by-*nrhs*. The solution *X* overwrites *B*.

The singular values of *A* smaller than *rcond* times the largest singular value are treated as zero in solving the least squares problem; in this case a minimum norm solution is returned. The actual singular values are returned in *d* in ascending order.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2.

It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

Input Parameters

uplo

CHARACTER*1.

If *uplo* = 'U', *d* and *e* define an upper bidiagonal matrix.

If *uplo* = 'L', *d* and *e* define a lower bidiagonal matrix.

smlsiz

INTEGER. The maximum size of the subproblems at the bottom of the computation tree.

n

INTEGER. The dimension of the bidiagonal matrix.

n≥ 0.

nrhs

INTEGER. The number of columns of *B*. Must be at least 1.

d

REAL for slalsd/clalsd

DOUBLE PRECISION for dlalsd/zlalsd

Array, DIMENSION (*n*). On entry, *d* contains the main diagonal of the bidiagonal matrix.

e

REAL for slalsd/clalsd

DOUBLE PRECISION for dlalsd/zlalsd

Array, DIMENSION (*n*-1). Contains the super-diagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.

b

REAL for slalsd

DOUBLE PRECISION for dlalsd

COMPLEX for clalsd

DOUBLE COMPLEX for zlalsd

Array, DIMENSION (*ldb*,*nrhs*).

On input, *b* contains the right hand sides of the least squares problem. On output, *b* contains the solution X.

ldb

INTEGER. The leading dimension of *b* in the calling subprogram. Must be at least max(1,*n*).

rcond

REAL for slalsd/clalsd

DOUBLE PRECISION for dlalsd/zlalsd

The singular values of *A* less than or equal to *rcond* times the largest singular value are treated as zero in solving the least squares problem. If *rcond* is negative, machine precision is used instead. For example, for the least squares problem diag(*S*)*X=*B*, where diag(*S*) is a diagonal matrix of singular values, the solution is *X*(*i*)=*B*(*i*)/*S*(*i*) if *S*(*i*) is greater than *rcond* *max(*S*), and *X*(*i*)=0 if *S*(*i*) is less than or equal to *rcond* *max(*S*).

rank

INTEGER. The number of singular values of *A* greater than *rcond* times the largest singular value.

work

REAL for slalsd

DOUBLE PRECISION for dlalsd

COMPLEX for clalsd

DOUBLE COMPLEX for zlalsd

Workspace array.

DIMENSION for real flavors at least

(9*n*+2*n**smlsiz*+8*n**nlvl*+*n**nrhs*+(*smlsiz*+1)^2),

where

nlvl = max(0, int($\log_2$(*n*/(*smlsiz*+1))) + 1).

DIMENSION for complex flavors is (*n**nrhs*).

rwork

REAL for clalsd

DOUBLE PRECISION for zlalsd

Workspace array, used with complex flavors only.

DIMENSION at least (9*n* + 2*n**smlsiz* + 8*n**nlvl* + 3**mlsiz**nrhs* + (*smlsiz*+1)^2),

where

nlvl = max(0, int($\log_2$(min(*m*,*n*)/(*smlsiz*+1))) + 1).

iwork

INTEGER.

Workspace array of DIMENSION(3*n**nlvl* + 11*n*).

Output Parameters

d

On exit, if *info* = 0, *d* contains singular values of the bidiagonal matrix.

e	On exit, destroyed.
b	On exit, *b* contains the solution *X*.
info	INTEGER. If *info* = 0: successful exit. If *info* = -*i* < 0, the *i*-th argument had an illegal value. If *info* > 0: The algorithm failed to compute a singular value while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*,*n*+1).

?lamrg

Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in acsending order.

Syntax

```
call slamrg( n1, n2, a, strd1, strd2, index )
call dlamrg( n1, n2, a, strd1, strd2, index )
```

Include Files

- mkl.fi

Description

The routine creates a permutation list which will merge the elements of *a* (which is composed of two independently sorted sets) into a single set which is sorted in ascending order.

Input Parameters

n1, *n2*	INTEGER. These arguments contain the respective lengths of the two sorted lists to be merged.
a	REAL for slamrg DOUBLE PRECISION for dlamrg. Array, DIMENSION (*n1*+*n2*). The first *n1* elements of *a* contain a list of numbers which are sorted in either ascending or descending order. Likewise for the final *n2* elements.
strd1, *strd2*	INTEGER. These are the strides to be taken through the array *a*. Allowable strides are 1 and -1. They indicate whether a subset of *a* is sorted in ascending (*strdx* = 1) or descending (*strdx* = -1) order.

Output Parameters

index	INTEGER. Array, DIMENSION (*n1*+*n2*).

On exit, this array will contain a permutation such that if `b(i) = a(index(i))` for `i=1, n1+n2`, then *b* will be sorted in ascending order.

?lamswlq

Multiplies a general real matrix by a real orthogonal matrix defined as the product of blocked elementary reflectors computed by short wide LQ factorization.

```
call slamswlq(side, trans, m, n, k, mb, nb, a, lda, t, ldt, c, ldc, work, lwork, info)
call dlamswlq(side, trans, m, n, k, mb, nb, a, lda, t, ldt, c, ldc, work, lwork, info)
call clamswlq(side, trans, m, n, k, mb, nb, a, lda, t, ldt, c, ldc, work, lwork, info)
call zlamswlq(side, trans, m, n, k, mb, nb, a, lda, t, ldt, c, ldc, work, lwork, info)
```

Description

`?lamswlq` overwrites the general real *m*-by-*n* matrix *C* with

	side = 'L'	*side* = 'R'
trans = 'N'	*Q***C*	*C***Q*
trans = 'T'	Q^T**C*	*C**Q^T
trans = 'C'	Q^H**C*	*C**Q^H

where *Q* is a real orthogonal matrix defined as the product of blocked elementary reflectors computed by short wide LQ factorization (`?laswlq`).

Short-Wide LQ (SWLQ) performs LQ by a sequence of orthogonal transformations, representing *Q* as a product of other orthogonal matrices: *Q* = *Q*(1) * *Q*(2) * . . . * *Q*(*k*), where each *Q*(*i*) zeros out upper diagonal entries of a block of *nb* rows of *A*:

Q(1) zeros out the upper diagonal entries of rows 1:*nb* of *A*,

Q(2) zeros out the bottom *mb*-*n* rows of rows [1:*m*, *nb* + 1:2**nb* - *m*] of *A*,

Q(3) zeros out the bottom *mb*-*n* rows of rows [1:*m*, 2**nb*-*m* + 1:3**nb* - 2**m*] of *A* . . . .

Q(1) is computed by `gelqt`, which represents *Q*(1) by Householder vectors stored under the diagonal of rows 1:*mb* of *A*, and by upper triangular block reflectors, stored in array *t*(1:*ldt*, 1:*n*). For more information, see `gelqt`.

Q(*i*) for *i* > 1 is computed by `tplqt`, which represents *Q*(*i*) by Householder vectors stored in columns [(*i* - 1)*(*nb* - *m*) + *m* + 1:*i**(*nb* - *m*) + *m*] of *A*, and by upper triangular block reflectors, stored in array *t*(1:*ldt*, (*i* - 1)**m* + 1:*i***m*). The last *Q*(*k*) may use fewer rows. For more information see Further Details in `tplqt`. For more details of the overall algorithm, see [DEMMEL12].

Input Parameters

side — CHARACTER*1.

If *side* = 'L': apply op(*Q*) from the left;

if *side* = 'R': apply op(*Q*) from the right.

trans — CHARACTER*1.

If *trans* = 'N': No transpose, op(*Q*) = *Q*;

if *trans* = 'T': Transpose, op(*Q*) = Q^T;

if *trans*= 'C': Transpose, op(*Q*) = Q^H.

m — INTEGER. The number of rows of the matrix *C*. *m*≥ 0.

n — INTEGER. The number of columns of the matrix *C*. *n*≥*m*.

k — INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. *m*≥*k*≥ 0;

mb — INTEGER. The row block size to be used in the blocked QR. *m*≥*mb*≥ 1

nb — INTEGER. The block size to be used in the blocked QR. *nb* > *m*.

a — REAL for slamswlq

DOUBLE PRECISION for dlamswlq

COMPLEX for clamswlq

COMPLEX*16 for zlamswlq

Array of size (*lda*, *m*) if *side* = 'L' or (*lda*, *n*) if *side* = 'R'. The *i*-th row must contain the vector which defines the blocked elementary reflector *H*(*i*), for *i* = 1, 2, ..., *k*, as returned by ?laswlq in the first *k* rows of its array argument *a*.

lda — INTEGER. The leading dimension of the array *a*.*lda*≥ max(1, *k*).

t — REAL for slamswlq

DOUBLE PRECISION for dlamswlq

COMPLEX for clamswlq

COMPLEX*16 for zlamswlq

Array of size (*m* * Number of blocks(ceiling(*n* - *k*/*nb* - *k*))), The blocked upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks as described previously.

ldt — INTEGER. The leading dimension of the array *t*. *ldt*≥*mb*.

c — REAL for slamswlq

DOUBLE PRECISION for dlamswlq

COMPLEX for clamswlq

COMPLEX*16 for zlamswlq

Array of size (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.

ldc — INTEGER. The leading dimension of the array *c*. *ldc*≥ max(1,*m*).

lwork — INTEGER. The size of the array *work*. If *side* = 'L', *lwork*≥ max(1, *nb*) * *mb*; if *side* = 'R', *lwork*≥ max(1, *m*) * *mb*. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

c	On exit, *c* is overwritten by op(*Q*)***C* or *C**op(*Q*).
work	REAL for slamswlq DOUBLE PRECISION for dlamswlq COMPLEX for clamswlq COMPLEX*16 for zlamswlq Workspace array of size (max(1, *lwork*)).
info	INTEGER. *info* = 0: successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?lamtsqr

Multiplies a general matrix by the product of blocked elementary reflectors computed by tall skinny QR factorization (?latsqr)

```
call slamtsqr(side, trans, m, n, k, mb, nb, a, lda, t, ldt, c, ldc, work, lwork, info)
call dlamtsqr(side, trans, m, n, k, mb, nb, a, lda, t, ldt, c, ldc, work, lwork, info)
call clamtsqr(side, trans, m, n, k, mb, nb, a, lda, t, ldt, c, ldc, work, lwork, info)
call zlamtsqr(side, trans, m, n, k, mb, nb, a, lda, t, ldt, c, ldc, work, lwork, info)
```

Description

?lamtsqr overwrites the general real or complex*m*-by-*n* matrix *C* with

	side = 'L'	*side* = 'R'
trans = 'N'	*Q***C*	*C***Q*
trans = 'T'	Q^T**C*	*C**Q^T
trans = 'C'	Q^H**C*	*C**Q^H

where *Q* is a real orthogonal matrix defined as the product of blocked elementary reflectors computed by tall skinny QR factorization (?latsqr). Tall-Skinny QR (TSQR) performs QR by a sequence of orthogonal transformations, representing *Q* as a product of other orthogonal matrices

Q = *Q*(1) * *Q*(2) * . . . * *Q*(*k*)

where each *Q*(*i*) zeros out subdiagonal entries of a block of *mb* rows of *A*:

Q(1) zeros out the subdiagonal entries of rows 1:*mb* of *A*,

Q(2) zeros out the bottom *mb*-*n* rows of rows [1:*n*, *mb* + 1:2**mb* - *n*] of *A*,

Q(3) zeros out the bottom *mb*-*n* rows of rows [1:*n*, 2**mb* - *n* + 1:3**mb* - 2**n*] of *A* . . . .

Q(1) is computed by geqrt, which represents *Q*(1) by Householder vectors stored under the diagonal of rows 1:*mb* of *a*, and by upper triangular block reflectors, stored in array *t*(1:*ldt*, 1:*n*). For more information, see geqrt.

$Q(i)$ for $i > 1$ is computed by `tpqrt`, which represents $Q(i)$ by Householder vectors stored in rows [(*i* - 1)*(*mb* - *n*) + *n* + 1:*i**(*mb* - *n*) + *n*] of *a*, and by upper triangular block reflectors, stored in array *t*(1:*ldt*, (*i* - 1)**n* + 1:*i***n*). The last $Q(k)$ may use fewer rows. For more information, see `tpqrt`. For more details of the overall algorithm, see [DEMMEL12].

Input Parameters

side
`CHARACTER*1`.
If *side* = 'L': apply op(Q) from the left;
if *side* = 'R': apply op(Q) from the right.

trans
`CHARACTER*1`.
If *trans* = 'N': No transpose, op(Q) = Q;
if *trans* = 'T': Transpose, op(Q) = Q^T;
if *trans* = 'C': Transpose, op(Q) = Q^H.

m
`INTEGER`. The number of rows of the matrix C. $m \geq 0$.

n
`INTEGER`. The number of columns of the matrix C. $m \geq n \geq 0$.

k
`INTEGER`. The number of elementary reflectors whose product defines the matrix Q. $n \geq k \geq 0$;

mb
`INTEGER`. The block size to be used in the blocked QR. *mb* > *n*. (Must be the same as in `?latsqr`)

nb
`INTEGER`. The column block size to be used in the blocked QR. $n \geq nb \geq 1$.

a
`REAL` for `slamtsqr`
`DOUBLE PRECISION` for `dlamtsqr`
`COMPLEX` for `clamtsqr`
`COMPLEX*16` for `zlamtsqr`
Array of size (*lda*, *k*). The *i*-th column must contain the vector which defines the blocked elementary reflector $H(i)$, for $i = 1,2,...,k$, as returned by `?latsqr` in the first *k* columns of its array argument *a*.

lda
`INTEGER`. The leading dimension of the array *a*.
If *side* = 'L', $lda \geq \max(1, m)$;
if *side* = 'R', $lda \geq \max(1, n)$.

t
`REAL` for `slamtsqr`
`DOUBLE PRECISION` for `dlamtsqr`
`COMPLEX` for `clamtsqr`
`COMPLEX*16` for `zlamtsqr`
Array of size (*n* * Number of blocks(ceiling(*m*-*k*/*mb*-*k*))). The blocked upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks, as described previously.

ldt
`INTEGER`. The leading dimension of the array *t*. $ldt \geq nb$.

c	REAL for slamtsqr DOUBLE PRECISION for dlamtsqr COMPLEX for clamtsqr COMPLEX*16 for zlamtsqr Array of size (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.
ldc	INTEGER. The leading dimension of the array *c*. *ldc*≥ max(1, *m*).
lwork	INTEGER. The dimension of the array *work*. If *side* = 'L', *lwork*≥ max(1, *n*)**nb*; if *side* = 'R', *lwork*≥ max(1, *mb*)**nb*. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

c	On exit, *c* is overwritten by op(*Q*)**C* or *C**op(*Q*).
work	REAL for slamtsqr DOUBLE PRECISION for dlamtsqr COMPLEX for clamtsqr COMPLEX*16 for zlamtsqr Workspace array of size (max(1, *lwork*)).
info	INTEGER. *info* = 0: successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?laneg

*Computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal T-sigma*I = L*D*L^T.*

Syntax

```
value = slaneg( n, d, lld, sigma, pivmin, r )
value = dlaneg( n, d, lld, sigma, pivmin, r )
```

Include Files

- mkl.fi

Description

The routine computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal $T-sigma*I = L*D*L^T$. This implementation works directly on the factors without forming the tridiagonal matrix *T*. The Sturm count is also the number of eigenvalues of *T* less than *sigma*. This routine is called from ?larb. The current routine does not use the *pivmin* parameter but rather requires IEEE-754

propagation of infinities and NaNs (NaN stands for 'Not A Number'). This routine also has no input range restrictions but does require default exception handling such that *x*/0 produces Inf when *x* is non-zero, and Inf/Inf produces NaN. (For more information see [Marques06]).

Input Parameters

n	INTEGER. The order of the matrix.
d	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (*n*). Contains *n* diagonal elements of the matrix *D*.
lld	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (*n*-1). Contains (*n*-1) elements *L*(i)**L*(i)**D*(i).
sigma	REAL for slaneg DOUBLE PRECISION for dlaneg Shift amount in *T-sigma*I = L*D*L**T*.
pivmin	REAL for slaneg DOUBLE PRECISION for dlaneg The minimum pivot in the Sturm sequence. May be used when zero pivots are encountered on non-IEEE-754 architectures.
r	INTEGER. The twist index for the twisted factorization that is used for the negcount.

Output Parameters

value	INTEGER. The number of negative pivots encountered while factoring.

?langb

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.

Syntax

```
val = slangb( norm, n, kl, ku, ab, ldab, work )
val = dlangb( norm, n, kl, ku, ab, ldab, work )
val = clangb( norm, n, kl, ku, ab, ldab, work )
val = zlangb( norm, n, kl, ku, ab, ldab, work )
```

Include Files

- mkl.fi

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* band matrix *A*, with *kl* sub-diagonals and *ku* super-diagonals.

Input Parameters

norm — CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

n — INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?langb is set to zero.

kl — INTEGER. The number of sub-diagonals of the matrix *A*. $kl \geq 0$.

ku — INTEGER. The number of super-diagonals of the matrix *A*. $ku \geq 0$.

ab — REAL for slangb

DOUBLE PRECISION for dlangb

COMPLEX for clangb

DOUBLE COMPLEX for zlangb

Array, DIMENSION (*ldab*,*n*).

The band matrix *A*, stored in rows 1 to *kl*+*ku*+1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

ab(*ku*+1+*i*-*j*,*j*) = *a*(*i*,*j*)

for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$.

ldab — INTEGER. The leading dimension of the array *ab*.

$ldab \geq kl+ku+1$.

work — REAL for slangb/clangb

DOUBLE PRECISION for dlangb/zlangb

Workspace array, DIMENSION (max(1,*lwork*)), where

$lwork \geq n$ when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slangb/clangb

DOUBLE PRECISION for dlangb/zlangb

Value returned by the function.

?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.

Syntax

```
val = slange( norm, m, n, a, lda, work )
val = dlange( norm, m, n, a, lda, work )
val = clange( norm, m, n, a, lda, work )
val = zlange( norm, m, n, a, lda, work )
```

Include Files

- mkl.fi

Description

The function ?lange returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex matrix *A*.

Input Parameters

The data types are given for the Fortran interface.

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

m

INTEGER. The number of rows of the matrix *A*.

m ≥ 0. When *m* = 0, ?lange is set to zero.

n

INTEGER. The number of columns of the matrix *A*.

n ≥ 0. When *n* = 0, ?lange is set to zero.

a

REAL for slange

DOUBLE PRECISION for dlange

COMPLEX for clange

DOUBLE COMPLEX for zlange

Array, DIMENSION (*lda*,*n*).

Array *a* contains the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.

.

work REAL for slange and clange.

DOUBLE PRECISION for dlange and zlange.

Workspace array, DIMENSION max(1,*lwork*), where *lwork*≥*m* when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val REAL for slange/clange

DOUBLE PRECISION for dlange/zlange

Value returned by the function.

?langt

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.

Syntax

```
val = slangt( norm, n, dl, d, du )
val = dlangt( norm, n, dl, d, du )
val = clangt( norm, n, dl, d, du )
val = zlangt( norm, n, dl, d, du )
```

Include Files

- mkl.fi

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex tridiagonal matrix *A*.

Input Parameters

norm CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

n	INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?langt is set to zero.
dl, *d*, *du*	REAL for slangt DOUBLE PRECISION for dlangt COMPLEX for clangt DOUBLE COMPLEX for zlangt Arrays: *dl* (*n*-1), *d* (*n*), *du* (*n*-1). The array *dl* contains the (*n*-1) sub-diagonal elements of *A*. The array *d* contains the diagonal elements of *A*. The array *du* contains the (*n*-1) super-diagonal elements of *A*.

Output Parameters

val	REAL for slangt/clangt DOUBLE PRECISION for dlangt/zlangt Value returned by the function.

?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.

Syntax

val = slanhs(*norm*, *n*, *a*, *lda*, *work*)

val = dlanhs(*norm*, *n*, *a*, *lda*, *work*)

val = clanhs(*norm*, *n*, *a*, *lda*, *work*)

val = zlanhs(*norm*, *n*, *a*, *lda*, *work*)

Include Files

- mkl.fi

Description

The function ?lanhs returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hessenberg matrix *A*.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm	CHARACTER*1. Specifies the value to be returned by the routine as described above.
n	INTEGER. The order of the matrix *A*. *n* ≥ 0. When *n* = 0, ?lanhs is set to zero.
a	REAL for slanhs DOUBLE PRECISION for dlanhs COMPLEX for clanhs DOUBLE COMPLEX for zlanhs Array, DIMENSION (*lda*,*n*). The *n*-by-*n* upper Hessenberg matrix *A*; the part of *A* below the first sub-diagonal is not referenced.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(*n*,1).
work	REAL for slanhs and clanhs. DOUBLE PRECISION for dlange and zlange. Workspace array, DIMENSION(max(1,*lwork*)), where *lwork* ≥ *n* when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val	REAL for slanhs/clanhs DOUBLE PRECISION for dlanhs/zlanhs Value returned by the function.

?lansb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.

Syntax

```
val = slansb( norm, uplo, n, k, ab, ldab, work )
val = dlansb( norm, uplo, n, k, ab, ldab, work )
val = clansb( norm, uplo, n, k, ab, ldab, work )
val = zlansb( norm, uplo, n, k, ab, ldab, work )
```

Include Files

- mkl.fi

Description

The function ?lansb returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* real/complex symmetric band matrix *A*, with *k* super-diagonals.

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the band matrix *A* is supplied. If *uplo* = 'U': upper triangular part is supplied; If *uplo* = 'L': lower triangular part is supplied.

n

INTEGER. The order of the matrix *A*. $n \geq 0$.

When *n* = 0, ?lansb is set to zero.

k

INTEGER. The number of super-diagonals or sub-diagonals of the band matrix *A*. $k \geq 0$.

ab

REAL for slansb

DOUBLE PRECISION for dlansb

COMPLEX for clansb

DOUBLE COMPLEX for zlansb

Array, DIMENSION (*ldab*,*n*).

The upper or lower triangle of the symmetric band matrix *A*, stored in the first *k*+1 rows of *ab*. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

if *uplo* = 'U', *ab*(*k*+1+*i*-*j*,*j*) = *a*(*i*,*j*)

for max(1,*j*-*k*) ≤ *i* ≤ *j*;

if *uplo* = 'L', *ab*(1+*i*-*j*,*j*) = *a*(*i*,*j*) for $j \leq i \leq \min(n, j+k)$.

ldab

INTEGER. The leading dimension of the array *ab*.

$ldab \geq k+1$.

work

REAL for slansb and clansb.

DOUBLE PRECISION for dlansb and zlansb.

Workspace array, DIMENSION(max(1,*lwork*)), where

$lwork \geq n$ when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slansb/clansb

DOUBLE PRECISION for dlansb/zlansb

Value returned by the function.

?lanhb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.

Syntax

```
val = clanhb( norm, uplo, n, k, ab, ldab, work )
val = zlanhb( norm, uplo, n, k, ab, ldab, work )
```

Include Files

- mkl.fi

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* Hermitian band matrix *A*, with *k* super-diagonals.

Input Parameters

norm — CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo — CHARACTER*1.

Specifies whether the upper or lower triangular part of the band matrix *A* is supplied.

If *uplo* = 'U': upper triangular part is supplied;

If *uplo* = 'L': lower triangular part is supplied.

n — INTEGER. The order of the matrix *A*. $n \geq 0$. When *n* = 0, ?lanhb is set to zero.

k — INTEGER. The number of super-diagonals or sub-diagonals of the band matrix *A*.

$k \geq 0$.

ab — COMPLEX for clanhb.

DOUBLE COMPLEX for zlanhb.

Array, DIMENSION (*ldaB*,*n*). The upper or lower triangle of the Hermitian band matrix *A*, stored in the first *k*+1 rows of *ab*. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

if *uplo* = 'U', *ab*(*k*+1+*i*-*j*,*j*) = *a*(*i*,*j*)

for max(1,*j*-*k*) ≤ *i* ≤ *j*;

if *uplo* = 'L', *ab*(1+*i*-*j*,*j*) = *a*(*i*,*j*) for *j* ≤ *i* ≤ min(*n*,*j*+*k*).

Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero.

ldab — INTEGER. The leading dimension of the array *ab*. *ldab*≥*k*+1.

work — REAL for clanhb.

DOUBLE PRECISION for zlanhb.

Workspace array, DIMENSIONmax(1, *lwork*), where

lwork≥*n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slanhb/clanhb

DOUBLE PRECISION for dlanhb/zlanhb

Value returned by the function.

?lansp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.

Syntax

```
val = slansp( norm, uplo, n, ap, work )
val = dlansp( norm, uplo, n, ap, work )
val = clansp( norm, uplo, n, ap, work )
val = zlansp( norm, uplo, n, ap, work )
```

Include Files

- mkl.fi

Description

The function ?lansp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix *A*, supplied in packed form.

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is supplied.

If *uplo* = 'U': Upper triangular part of *A* is supplied

If *uplo* = 'L': Lower triangular part of *A* is supplied.

n

INTEGER. The order of the matrix *A*. $n \geq 0$. When

$n = 0$, ?lansp is set to zero.

ap

REAL for slansp

DOUBLE PRECISION for dlansp

COMPLEX for clansp

DOUBLE COMPLEX for zlansp

Array, DIMENSION ($n(n+1)/2$).

The upper or lower triangle of the symmetric matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;

if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

work

REAL for slansp and clansp.

DOUBLE PRECISION for dlansp and zlansp.

Workspace array, DIMENSION(max(1, *lwork*)), where

$lwork \geq n$ when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val

REAL for slansp/clansp

DOUBLE PRECISION for dlansp/zlansp

Value returned by the function.

?lanhp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.

Syntax

```
val = clanhp( norm, uplo, n, ap, work )
val = zlanhp( norm, uplo, n, ap, work )
```

Include Files

- mkl.fi

Description

The function ?lanhp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix *A*, supplied in packed form.

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is supplied.

If *uplo* = 'U': Upper triangular part of *A* is supplied

If *uplo* = 'L': Lower triangular part of *A* is supplied.

n

INTEGER. The order of the matrix *A*.

$n \geq 0$. When $n = 0$, ?lanhp is set to zero.

ap

COMPLEX for clanhp.

DOUBLE COMPLEX for zlanhp.

Array, DIMENSION ($n(n+1)/2$). The upper or lower triangle of the Hermitian matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;

if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

work

REAL for clanhp.

DOUBLE PRECISION for zlanhp.

Workspace array, DIMENSION(max(1,*lwork*)), where

lwork≥*n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val

REAL for clanhp.

DOUBLE PRECISION for zlanhp.

Value returned by the function.

?lanst/?lanht

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.

Syntax

```
val = slanst( norm, n, d, e )
val = dlanst( norm, n, d, e )
val = clanht( norm, n, d, e )
val = zlanht( norm, n, d, e )
```

Include Files

- mkl.fi

Description

The functions ?lanst/?lanht return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or a complex Hermitian tridiagonal matrix *A*.

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

n

INTEGER. The order of the matrix *A*.

n≥ 0. When *n* = 0, ?lanst/?lanht is set to zero.

d — REAL for slanst/clanht

DOUBLE PRECISION for dlanst/zlanht

Array, DIMENSION (*n*). The diagonal elements of *A*.

e — REAL for slanst

DOUBLE PRECISION for dlanst

COMPLEX for clanht

DOUBLE COMPLEX for zlanht

Array, DIMENSION (*n*-1).

The (*n*-1) sub-diagonal or super-diagonal elements of *A*.

Output Parameters

val — REAL for slanst/clanht

DOUBLE PRECISION for dlanst/zlanht

Value returned by the function.

?lansy

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.

Syntax

```
val = slansy( norm, uplo, n, a, lda, work )
val = dlansy( norm, uplo, n, a, lda, work )
val = clansy( norm, uplo, n, a, lda, work )
val = zlansy( norm, uplo, n, a, lda, work )
```

Include Files

- mkl.fi

Description

The function ?lansy returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix *A*.

Input Parameters

The data types are given for the Fortran interface.

norm — CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo — CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is to be referenced.

= 'U': Upper triangular part of *A* is referenced.

= 'L': Lower triangular part of *A* is referenced

n — INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?lansy is set to zero.

a — REAL for slansy

DOUBLE PRECISION for dlansy

COMPLEX for clansy

DOUBLE COMPLEX for zlansy

Array, size (*lda*,*n*). The symmetric matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. The leading dimension of the array *a*.

lda ≥ max(*n*,1).

work — REAL for slansy and clansy.

DOUBLE PRECISION for dlansy and zlansy.

Workspace array, DIMENSION(max(1,*lwork*)), where

lwork ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slansy/clansy

DOUBLE PRECISION for dlansy/zlansy

Value returned by the function.

?lanhe

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.

Syntax

val = clanhe(*norm*, *uplo*, *n*, *a*, *lda*, *work*)

val = zlanhe(*norm*, *uplo*, *n*, *a*, *lda*, *work*)

Include Files

- mkl.fi

Description

The function ?lanhe returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix *A*.

Input Parameters

The data types are given for the Fortran interface.

norm — CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo — CHARACTER*1.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is to be referenced.

= 'U': Upper triangular part of *A* is referenced.

= 'L': Lower triangular part of *A* is referenced

n — INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?lanhe is set to zero.

a — COMPLEX for clanhe.

DOUBLE COMPLEX for zlanhe.

Array, size (*lda*,*n*). The Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. The leading dimension of the array *a*.

lda ≥ max(*n*,1).

work — REAL for clanhe.

DOUBLE PRECISION for zlanhe.

Workspace array, DIMENSION(max(1,*lwork*)), where

lwork≥*n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val

REAL for clanhe.

DOUBLE PRECISION for zlanhe.

Value returned by the function.

?lantb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.

Syntax

```
val = slantb( norm, uplo, diag, n, k, ab, ldab, work )
val = dlantb( norm, uplo, diag, n, k, ab, ldab, work )
val = clantb( norm, uplo, diag, n, k, ab, ldab, work )
val = zlantb( norm, uplo, diag, n, k, ab, ldab, work )
```

Include Files

- mkl.fi

Description

The function ?lantb returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* triangular band matrix *A*, with (*k* + 1) diagonals.

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo

CHARACTER*1.

Specifies whether the matrix *A* is upper or lower triangular.

= 'U': Upper triangular

= 'L': Lower triangular.

diag CHARACTER*1.

Specifies whether or not the matrix *A* is unit triangular.

= 'N': Non-unit triangular

= 'U': Unit triangular.

n INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?lantb is set to zero.

k INTEGER. The number of super-diagonals of the matrix *A* if *uplo* = 'U', or the number of sub-diagonals of the matrix *A* if *uplo* = 'L'. $k \geq 0$.

ab REAL for slantb

DOUBLE PRECISION for dlantb

COMPLEX for clantb

DOUBLE COMPLEX for zlantb

Array, DIMENSION (*ldab*,*n*). The upper or lower triangular band matrix *A*, stored in the first *k*+1 rows of *ab*.

The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

if *uplo* = 'U', $ab(k+1+i-j,j) = a(i,j)$ for $\max(1,j-k) \leq i \leq j$;

if *uplo* = 'L', $ab(1+i-j,j) = a(i,j)$ for $j \leq i \leq \min(n,j+k)$.

Note that when *diag* = 'U', the elements of the array *ab* corresponding to the diagonal elements of the matrix *A* are not referenced, but are assumed to be one.

ldab INTEGER. The leading dimension of the array *ab*.

$ldab \geq k+1$.

work REAL for slantb and clantb.

DOUBLE PRECISION for dlantb and zlantb.

Workspace array, DIMENSION(max(1,*lwork*)), where

$lwork \geq n$ when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

val REAL for slantb/clantb.

DOUBLE PRECISION for dlantb/zlantb.

Value returned by the function.

?lantp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.

Syntax

```
val = slantp( norm, uplo, diag, n, ap, work )
val = dlantp( norm, uplo, diag, n, ap, work )
val = clantp( norm, uplo, diag, n, ap, work )
val = zlantp( norm, uplo, diag, n, ap, work )
```

Include Files

- mkl.fi

Description

The function ?lantp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix *A*, supplied in packed form.

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo

CHARACTER*1.

Specifies whether the matrix *A* is upper or lower triangular.

= 'U': Upper triangular

= 'L': Lower triangular.

diag

CHARACTER*1.

Specifies whether or not the matrix *A* is unit triangular.

= 'N': Non-unit triangular

= 'U': Unit triangular.

n

INTEGER. The order of the matrix *A*.

$n \geq 0$. When $n = 0$, ?lantp is set to zero.

ap

REAL for slantp

DOUBLE PRECISION for dlantp

COMPLEX for clantp

DOUBLE COMPLEX for zlantp

Array, DIMENSION ($n(n+1)/2$).

The upper or lower triangular matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', AP(i + (*j*-1)*j*/2) =*a*(*i*,*j*) for 1≤ *i*≤ *j*;

if *uplo* = 'L', *ap*(*i* + (*j*-1)(2*n*-*j*)/2) = *a*(*i*,*j*) for *j*≤ *i*≤ *n*.

Note that when *diag* = 'U', the elements of the array *ap* corresponding to the diagonal elements of the matrix *A* are not referenced, but are assumed to be one.

work

REAL for slantp and clantp.

DOUBLE PRECISION for dlantp and zlantp.

Workspace array, DIMENSION(max(1,*lwork*)), where *lwork*≥*n* when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

val

REAL for slantp/clantp.

DOUBLE PRECISION for dlantp/zlantp.

Value returned by the function.

?lantr

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.

Syntax

val = slantr(*norm*, *uplo*, *diag*, *m*, *n*, *a*, *lda*, *work*)

val = dlantr(*norm*, *uplo*, *diag*, *m*, *n*, *a*, *lda*, *work*)

val = clantr(*norm*, *uplo*, *diag*, *m*, *n*, *a*, *lda*, *work*)

val = zlantr(*norm*, *uplo*, *diag*, *m*, *n*, *a*, *lda*, *work*)

Include Files

- mkl.fi

Description

The function ?lantr returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix *A*.

Input Parameters

The data types are given for the Fortran interface.

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo — CHARACTER*1.

Specifies whether the matrix *A* is upper or lower trapezoidal.

= 'U': Upper trapezoidal

= 'L': Lower trapezoidal.

Note that *A* is triangular instead of trapezoidal if *m* = *n*.

diag — CHARACTER*1.

Specifies whether or not the matrix *A* has unit diagonal.

= 'N': Non-unit diagonal

= 'U': Unit diagonal.

m — INTEGER. The number of rows of the matrix *A*. *m*≥ 0, and if *uplo* = 'U', *m* ≤ *n*.

When *m* = 0, ?lantr is set to zero.

n — INTEGER. The number of columns of the matrix *A*. *n*≥ 0, and if *uplo* = 'L', *n* ≤ *m*.

When *n* = 0, ?lantr is set to zero.

a — REAL for slantr

DOUBLE PRECISION for dlantr

COMPLEX for clantr

DOUBLE COMPLEX for zlantr

Array, DIMENSION (*lda*,*n*).

The trapezoidal matrix *A* (*A* is triangular if *m* = *n*).

If *uplo* = 'U', the leading *m*-by-*n* upper trapezoidal part of the array *a* contains the upper trapezoidal matrix, and the strictly lower triangular part of *A* is not referenced.

If *uplo* = 'L', the leading *m*-by-*n* lower trapezoidal part of the array *a* contains the lower trapezoidal matrix, and the strictly upper triangular part of *A* is not referenced. Note that when *diag* = 'U', the diagonal elements of *A* are not referenced and are assumed to be one.

lda — INTEGER. The leading dimension of the array *a*.

lda≥ max(*m*,1).

work — REAL for slantr/clantrp.

DOUBLE PRECISION for dlantr/zlantr.

Workspace array, DIMENSION(max(1,*lwork*)), where

lwork≥*m* when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

val	REAL for slantr/clantrp. DOUBLE PRECISION for dlantr/zlantr. Value returned by the function.

?lanv2

Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.

Syntax

```
call slanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
call dlanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
```

Include Files

- mkl.fi

Description

The routine computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} cs & -sn \\ sn & cs \end{bmatrix} \begin{bmatrix} aa & bb \\ cc & dd \end{bmatrix} \begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix}$$

where either

1. *cc* = 0 so that *aa* and *dd* are real eigenvalues of the matrix, or
2. *aa* = *dd* and *bb***cc* < 0, so that *aa*± sqrt(*bb***cc*) are complex conjugate eigenvalues.

The routine was adjusted to reduce the risk of cancellation errors, when computing real eigenvalues, and to ensure, if possible, that abs(*rt1r*) ≥ abs(*rt2r*).

Input Parameters

a, *b*, *c*, *d*	REAL for slanv2 DOUBLE PRECISION for dlanv2. On entry, elements of the input matrix.

Output Parameters

a, *b*, *c*, *d*	On exit, overwritten by the elements of the standardized Schur form.
rt1r, *rt1i*, *rt2r*, *rt2i*	REAL for slanv2 DOUBLE PRECISION for dlanv2. The real and imaginary parts of the eigenvalues. If the eigenvalues are a complex conjugate pair, *rt1i* > 0.

cs, *sn*	REAL for slanv2 DOUBLE PRECISION for dlanv2. Parameters of the rotation matrix.

?lapll

Measures the linear dependence of two vectors.

Syntax

```
call slapll( n, x, incx, Y, incy, ssmin )
call dlapll( n, x, incx, Y, incy, ssmin )
call clapll( n, x, incx, Y, incy, ssmin )
call zlapll( n, x, incx, Y, incy, ssmin )
```

Include Files

- mkl.fi

Description

Given two column vectors *x* and *y* of length *n*, let

A = (*xy*) be the *n*-by-2 matrix.

The routine ?lapll first computes the *QR* factorization of *A* as *A* = *Q***R* and then computes the SVD of the 2-by-2 upper triangular matrix *R*. The smaller singular value of *R* is returned in *ssmin*, which is used as the measurement of the linear dependency of the vectors *x* and *y*.

Input Parameters

n	INTEGER. The length of the vectors *x* and *y*.
x	REAL for slapll DOUBLE PRECISION for dlapll COMPLEX for clapll DOUBLE COMPLEX for zlapll Array, DIMENSION(1+(*n*-1)*incx*). On entry, *x* contains the *n*-vector x.
y	REAL for slapll DOUBLE PRECISION for dlapll COMPLEX for clapll DOUBLE COMPLEX for zlapll Array, DIMENSION (1+(*n*-1)*incy*). On entry, *y* contains the *n*-vector y.
incx	INTEGER. The increment between successive elements of x; *incx* > 0.

incy INTEGER. The increment between successive elements of y; *incy* > 0.

Output Parameters

x On exit, *x* is overwritten.

y On exit, *y* is overwritten.

ssmin REAL for slapll/clapll

DOUBLE PRECISION for dlapll/zlapll

The smallest singular value of the *n*-by-2 matrix *A* = (*xy*).

?lapmr

Rearranges rows of a matrix as specified by a permutation vector.

Syntax

```
call slapmr( forwrd, m, n, x, ldx, k )
call dlapmr( forwrd, m, n, x, ldx, k )
call clapmr( forwrd, m, n, x, ldx, k )
call zlapmr( forwrd, m, n, x, ldx, k )
call lapmr( x,k[,forwrd] )
```

Include Files

- mkl.fi

Description

The ?lapmr routine rearranges the rows of the *m*-by-*n* matrix *X* as specified by the permutation *k*(1),*k*(2),...,*k*(*m*) of the integers 1,...,*m*.

If *forwrd* = .TRUE., forward permutation:

X(*k*(*i*,*)) is moved to *X*(*i*,*) for *i*= 1,2,...,*m*.

If *forwrd* = .FALSE., backward permutation:

X(*i*,*) is moved to *X*(*k*(*i*,*)) for *i* = 1,2,...,*m*.

Input Parameters

The data types are given for the Fortran interface.

forwrd LOGICAL.

If *forwrd* = .TRUE., forward permutation.

If *forwrd* = .FALSE., backward permutation.

m INTEGER. The number of rows of the matrix *X*. *m*≥ 0.

n INTEGER. The number of columns of the matrix *X*. *n*≥ 0.

x REAL for slapmr

DOUBLE PRECISION for dlapmr

	COMPLEX for clapmr DOUBLE COMPLEX for zlapmr Array, size (*ldx*,*n*)On entry, the *m*-by-*n* matrix *X*.
ldx	INTEGER. The leading dimension of the array *X*, *ldx*≥ max(1,*m*).
k	INTEGER. Array, size (*m*). On entry, *k* contains the permutation vector and is used as internal workspace.

Output Parameters

x	On exit, *x* contains the permuted matrix *X*.
k	On exit, *k* is reset to its original value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ?lapmr interface are as follows:

x	Holds the matrix *X* of size (*n*, *n*).
k	Holds the vector of length *m*.
forwrd	Specifies the permutation. Must be '.TRUE.' or '.FALSE.'.

See Also

?lapmt

?lapmt

Performs a forward or backward permutation of the columns of a matrix.

Syntax

```
call slapmt( forwrd, m, n, x, ldx, k )
call dlapmt( forwrd, m, n, x, ldx, k )
call clapmt( forwrd, m, n, x, ldx, k )
call zlapmt( forwrd, m, n, x, ldx, k )
```

Include Files

- mkl.fi

Description

The routine ?lapmt rearranges the columns of the *m*-by-*n* matrix *X* as specified by the permutation *k*(1),*k*(2),...,*k*(*n*) of the integers 1,...,*n*.

If *forwrd*= .TRUE., forward permutation:

X(*,*k*(*j*)) is moved to *X*(*,*j*) for *j*=1,2,...,*n*.

If *forwrd* = .FALSE., backward permutation:

`X(*,j)` is moved to `X(*,k(j))` for `j = 1,2,...,n`.

Input Parameters

forwrd	LOGICAL. If *forwrd*= .TRUE., forward permutation If *forwrd* = .FALSE., backward permutation
m	INTEGER. The number of rows of the matrix *X*. *m*≥ 0.
n	INTEGER. The number of columns of the matrix *X*. *n*≥ 0.
x	REAL for slapmt DOUBLE PRECISION for dlapmt COMPLEX for clapmt DOUBLE COMPLEX for zlapmt Array, size (*ldx*,*n*). On entry, the *m*-by-*n* matrix *X*.
ldx	INTEGER. The leading dimension of the array *x*, *ldx*≥ max(1,*m*).
k	INTEGER. Array, size (*n*). On entry, *k* contains the permutation vector and is used as internal workspace.

Output Parameters

x	On exit, *x* contains the permuted matrix *X*.
k	On exit, *k* is reset to its original value.

See Also

?lapmr

?lapy2

Returns $sqrt(x^2+y^2)$.

Syntax

```
val = slapy2( x, y )
val = dlapy2( x, y )
```

Include Files

- mkl.fi

Description

The function ?lapy2 returns $sqrt(x^2+y^2)$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

The data types are given for the Fortran interface.

x, *y*	REAL for slapy2

`DOUBLE PRECISION` for `dlapy2`

Specify the input values *x* and *y*.

Output Parameters

val

`REAL` for `slapy2`

`DOUBLE PRECISION` for `dlapy2`.

Value returned by the function.

If *val*=`-1D0`, the first argument was NaN.

If *val*=`-2D0`, the second argument was NaN.

?lapy3

Returns $sqrt(x^2+y^2+z^2)$.

Syntax

```
val = slapy3( x, y, z )
val = dlapy3( x, y, z )
```

Include Files

- `mkl.fi`

Description

The function `?lapy3` returns $sqrt(x^2+y^2+z^2)$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

The data types are given for the Fortran interface.

x, *y*, *z*

`REAL` for `slapy3`

`DOUBLE PRECISION` for `dlapy3`

Specify the input values *x*, *y* and *z*.

Output Parameters

val

`REAL` for `slapy3`

`DOUBLE PRECISION` for `dlapy3`.

Value returned by the function.

If *val* = `-1D0`, the first argument was NaN.

If *val* = `-2D0`, the second argument was NaN.

If *val* = `-3D0`, the third argument was NaN.

?laqgb

Scales a general band matrix, using row and column scaling factors computed by `?gbequ`.

Syntax

```
call slaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call dlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call claqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call zlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a general *m*-by-*n* band matrix *A* with *kl* subdiagonals and *ku* superdiagonals using the row and column scaling factors in the vectors *r* and *c*.

Input Parameters

m

INTEGER. The number of rows of the matrix *A*. $m \geq 0$.

n

INTEGER. The number of columns of the matrix *A*. $n \geq 0$.

kl

INTEGER. The number of subdiagonals within the band of *A*. $kl \geq 0$.

ku

INTEGER. The number of superdiagonals within the band of *A*. $ku \geq 0$.

ab

REAL for slaqgb

DOUBLE PRECISION for dlaqgb

COMPLEX for claqgb

DOUBLE COMPLEX for zlaqgb

Array, DIMENSION (*ldab*,*n*). On entry, the matrix *A* in band storage, in rows 1 to *kl*+*ku*+1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows: $ab(ku+1+i-j,j) = A(i,j)$ for $\max(1,j-ku) \leq i \leq \min(m,j+kl)$.

ldab

INTEGER. The leading dimension of the array *ab*.

$lda \geq kl+ku+1$.

amax

REAL for slaqgb/claqgb

DOUBLE PRECISION for dlaqgb/zlaqgb

Absolute value of largest matrix entry.

r, *c*

REAL for slaqgb/claqgb

DOUBLE PRECISION for dlaqgb/zlaqgb

Arrays *r* (*m*), *c* (*n*). Contain the row and column scale factors for *A*, respectively.

rowcnd

REAL for slaqgb/claqgb

DOUBLE PRECISION for dlaqgb/zlaqgb

Ratio of the smallest $r(i)$ to the largest $r(i)$.

colcnd

REAL for slaqgb/claqgb

DOUBLE PRECISION for dlaqgb/zlaqgb

Ratio of the smallest $c(i)$ to the largest $c(i)$.

Output Parameters

ab

On exit, the equilibrated matrix, in the same storage format as *A*.

See *equed* for the form of the equilibrated matrix.

equed

CHARACTER*1.

Specifies the form of equilibration that was done.

If *equed* = 'N': No equilibration

If *equed* = 'R': Row equilibration, that is, *A* has been premultiplied by diag(*r*).

If *equed* = 'C': Column equilibration, that is, *A* has been postmultiplied by diag(*c*).

If *equed* = 'B': Both row and column equilibration, that is, *A* has been replaced by diag(*r*)*A*diag(*c*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If *rowcnd* < *thresh*, row scaling is done, and if *colcnd* < *thresh*, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, row scaling is done.

?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.

Syntax

```
call slaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call dlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call claqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call zlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a general *m*-by-*n* matrix *A* using the row and column scaling factors in the vectors *r* and *c*.

Input Parameters

m — INTEGER. The number of rows of the matrix *A*.

m ≥ 0.

n — INTEGER. The number of columns of the matrix *A*.

n ≥ 0.

a — REAL for slaqge

DOUBLE PRECISION for dlaqge

COMPLEX for claqge

DOUBLE COMPLEX for zlaqge

Array, DIMENSION (*lda*,*n*). On entry, the *m*-by-*n* matrix *A*.

lda — INTEGER. The leading dimension of the array *a*.

lda ≥ max(*m*,1).

r — REAL for slanqge/claqge

DOUBLE PRECISION for dlaqge/zlaqge

Array, DIMENSION (*m*). The row scale factors for A.

c — REAL for slanqge/claqge

DOUBLE PRECISION for dlaqge/zlaqge

Array, DIMENSION (*n*). The column scale factors for A.

rowcnd — REAL for slanqge/claqge

DOUBLE PRECISION for dlaqge/zlaqge

Ratio of the smallest *r*(*i*) to the largest *r*(*i*).

colcnd — REAL for slanqge/claqge

DOUBLE PRECISION for dlaqge/zlaqge

Ratio of the smallest *c*(*i*) to the largest *c*(*i*).

amax — REAL for slanqge/claqge

DOUBLE PRECISION for dlaqge/zlaqge

Absolute value of largest matrix entry.

Output Parameters

a — On exit, the equilibrated matrix.

See *equed* for the form of the equilibrated matrix.

equed — CHARACTER*1.

Specifies the form of equilibration that was done.

If *equed* = 'N': No equilibration

If *equed* = 'R': Row equilibration, that is, *A* has been premultiplied by diag(*r*).

If *equed* = 'C': Column equilibration, that is, *A* has been postmultiplied by diag(*c*).

If *equed* = 'B': Both row and column equilibration, that is, *A* has been replaced by diag(*r*)*A*diag(*c*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If *rowcnd* < *thresh*, row scaling is done, and if *colcnd* < *thresh*, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, row scaling is done.

?laqhb

Scales a Hermetian band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call claqhb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call zlaqhb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a Hermetian band matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo — CHARACTER*1.

Specifies whether the upper or lower triangular part of the band matrix *A* is stored.

If *uplo* = 'U': upper triangular.

If *uplo* = 'L': lower triangular.

n — INTEGER. The order of the matrix *A*.

n ≥ 0.

kd — INTEGER. The number of super-diagonals of the matrix *A* if *uplo* = 'U', or the number of sub-diagonals if *uplo* = 'L'.

kd ≥ 0.

ab — COMPLEX for claqhb

DOUBLE COMPLEX for zlaqhb

Array, DIMENSION (*ldab*,*n*). On entry, the upper or lower triangle of the band matrix *A*, stored in the first *kd*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

if *uplo* = 'U', *ab*(*kd*+1+*i*-*j*,*j*) = A(*i*,*j*) for max(1,*j*-*kd*) ≤ *i* ≤ *j*;

if *uplo* = 'L', *ab*(1+*i*-*j*,*j*) = A(*i*,*j*) for *j* ≤ *i* ≤ min(*n*,*j*+*kd*).

ldab — INTEGER. The leading dimension of the array *ab*.

ldab≥*kd*+1.

scond — REAL for claqsb

DOUBLE PRECISION for zlaqsb

Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax — REAL for claqsb

DOUBLE PRECISION for zlaqsb

Absolute value of largest matrix entry.

Output Parameters

ab — On exit, if *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization $A = U^H*U$ or $A = L*L^H$ of the band matrix *A*, in the same storage format as *A*.

s — REAL for claqsb

DOUBLE PRECISION for zlaqsb

Array, DIMENSION (*n*). The scale factors for A.

equed — CHARACTER*1.

Specifies whether or not equilibration was done.

If *equed* = 'N': No equilibration.

If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(*s*)*A*diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done.

The values *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqp2

Computes a QR factorization with column pivoting of the matrix block.

Syntax

```
call slaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call dlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
```

```
call claqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call zlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
```

Include Files

- mkl.fi

Description

The routine computes a *QR* factorization with column pivoting of the block `A(offset+1:m,1:n)`. The block `A(1:offset,1:n)` is accordingly pivoted, but not factorized.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. *m*≥ 0.
n	INTEGER. The number of columns of the matrix *A*. *n*≥ 0.
offset	INTEGER. The number of rows of the matrix *A* that must be pivoted but no factorized. *offset*≥ 0.
a	REAL for slaqp2 DOUBLE PRECISION for dlaqp2 COMPLEX for claqp2 DOUBLE COMPLEX for zlaqp2 Array, DIMENSION (*lda*,*n*). On entry, the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*m*).
jpvt	INTEGER. Array, DIMENSION (*n*). On entry, if *jpvt*(*i*) ≠ 0, the *i*-th column of *A* is permuted to the front of *A***P* (a leading column); if *jpvt*(*i*) = 0, the *i*-th column of *A* is a free column.
vn1, *vn2*	REAL for slaqp2/claqp2 DOUBLE PRECISION for dlaqp2/zlaqp2 Arrays, DIMENSION (*n*) each. Contain the vectors with the partial and exact column norms, respectively.
work	REAL for slaqp2 DOUBLE PRECISION for dlaqp2 COMPLEX for claqp2 DOUBLE COMPLEX for zlaqp2 Workspace array, DIMENSION (*n*).

Output Parameters

a	On exit, the upper triangle of block `A(offset+1:m,1:n)` is the triangular factor obtained; the elements in block `A(offset+1:m,1:n)` below the diagonal, together with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors. Block `A(1:offset,1:n)` has been accordingly pivoted, but not factorized.
jpvt	On exit, if `jpvt(i) = k`, then the *i*-th column of *A***P* was the *k*-th column of *A*.

tau	`REAL` for `slaqp2`
	`DOUBLE PRECISION` for `dlaqp2`
	`COMPLEX` for `claqp2`
	`DOUBLE COMPLEX` for `zlaqp2`
	Array, `DIMENSION(min(m,n))`.
	The scalar factors of the elementary reflectors.

vn1, *vn2*	Contain the vectors with the partial and exact column norms, respectively.

?laqps

Computes a step of QR factorization with column pivoting of a real m-by-n matrix A by using BLAS level 3.

Syntax

```
call slaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call dlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call claqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call zlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
```

Include Files

- `mkl.fi`

Description

The routine computes a step of *QR* factorization with column pivoting of a real *m*-by-*n* matrix *A* by using BLAS level 3. The routine tries to factorize *NB* columns from *A* starting from the row *offset*+1, and updates all of the matrix with BLAS level 3 routine `?gemm`.

In some cases, due to catastrophic cancellations, `?laqps` cannot factorize *NB* columns. Hence, the actual number of factorized columns is returned in *kb*.

Block `A(1:offset,1:n)` is accordingly pivoted, but not factorized.

Input Parameters

m	`INTEGER`. The number of rows of the matrix *A*. $m \geq 0$.
n	`INTEGER`. The number of columns of the matrix *A*. $n \geq 0$.

offset INTEGER. The number of rows of *A* that have been factorized in previous steps.

nb INTEGER. The number of columns to factorize.

a REAL for slaqps

DOUBLE PRECISION for dlaqps

COMPLEX for claqps

DOUBLE COMPLEX for zlaqps

Array, DIMENSION (*lda*,*n*).

On entry, the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.

lda ≥ max(1,*m*).

jpvt INTEGER. Array, DIMENSION (*n*).

If *jpvt*(*I*) = *k* then column *k* of the full matrix *A* has been permuted into position *i* in AP.

vn1, *vn2* REAL for slaqps/claqps

DOUBLE PRECISION for dlaqps/zlaqps

Arrays, DIMENSION (*n*) each. Contain the vectors with the partial and exact column norms, respectively.

auxv REAL for slaqps

DOUBLE PRECISION for dlaqps

COMPLEX for claqps

DOUBLE COMPLEX for zlaqps

Array, DIMENSION (*nb*). Auxiliary vector.

f REAL for slaqps

DOUBLE PRECISION for dlaqps

COMPLEX for claqps

DOUBLE COMPLEX for zlaqps

Array, DIMENSION (*ldf*,*nb*). For real flavors, matrix $F^T = L*Y^T*A$. For complex flavors, matrix $F^H = L*Y^H*A$.

ldf INTEGER. The leading dimension of the array *f*.

ldf ≥ max(1,*n*).

Output Parameters

kb INTEGER. The number of columns actually factorized.

a On exit, block *A*(*offset*+1:*m*,1:*kb*) is the triangular factor obtained and block *A*(1:*offset*,1:*n*) has been accordingly pivoted, but no factorized. The rest of the matrix, block *A*(*offset*+1:*m*,*kb*+1:*n*) has been updated.

jpvt	INTEGER array, DIMENSION (*n*). If *jpvt(I) = k* then column *k* of the full matrix *A* has been permuted into position *i* in AP.
tau	REAL for slaqps DOUBLE PRECISION for dlaqps COMPLEX for claqps DOUBLE COMPLEX for zlaqps Array, DIMENSION (*kb*). The scalar factors of the elementary reflectors.
vn1, *vn2*	The vectors with the partial and exact column norms, respectively.
auxv	Auxiliary vector.
f	Matrix *F′ = L*Y′*A*.

?laqr0

Computes the eigenvalues of a Hessenberg matrix, and optionally the marixes from the Schur decomposition.

Syntax

```
call slaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work, lwork, info )
call dlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work, lwork, info )
call claqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork, info )
call zlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork, info )
```

Include Files

- mkl.fi

Description

The routine computes the eigenvalues of a Hessenberg matrix *H*, and, optionally, the matrices *T* and *Z* from the Schur decomposition $H=Z*T*Z^H$, where *T* is an upper quasi-triangular/triangular matrix (the Schur form), and *Z* is the orthogonal/unitary matrix of Schur vectors.

Optionally *Z* may be postmultiplied into an input orthogonal/unitary matrix *Q* so that this routine can give the Schur factorization of a matrix *A* which has been reduced to the Hessenberg form *H* by the orthogonal/unitary matrix *Q*: $A = Q*H*Q^H = (QZ)*H*(QZ)^H$.

Input Parameters

wantt	LOGICAL. If *wantt* = .TRUE., the full Schur form T is required; If *wantt* = .FALSE., only eigenvalues are required.

wantz — LOGICAL.

If *wantz* = .TRUE., the matrix of Schur vectors *Z* is required;

If *wantz* = .FALSE., Schur vectors are not required.

n — INTEGER. The order of the Hessenberg matrix *H*. (*n* ≥ 0).

ilo, *ihi* — INTEGER.

It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*, and if *ilo* > 1 then *H*(*ilo*, *ilo*-1) = 0.

ilo and *ihi* are normally set by a previous call to cgebal, and then passed to cgehrd when the matrix output by cgebal is reduced to Hessenberg form. Otherwise, *ilo* and *ihi* should be set to 1 and *n*, respectively.

If *n* > 0, then 1 ≤ *ilo* ≤ *ihi* ≤ *n*.

If *n*=0, then *ilo*=1 and *ihi*=0

h — REAL for slaqr0

DOUBLE PRECISION for dlaqr0

COMPLEX for claqr0

DOUBLE COMPLEX for zlaqr0.

Array, DIMENSION (*ldh*, *n*), contains the upper Hessenberg matrix *H*.

ldh — INTEGER. The leading dimension of the array *h*. *ldh* ≥ max(1, *n*).

iloz, *ihiz* — INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE., 1 ≤ *iloz* ≤ *ilo*; *ihi* ≤ *ihiz* ≤ *n*.

z — REAL for slaqr0

DOUBLE PRECISION for dlaqr0

COMPLEX for claqr0

DOUBLE COMPLEX for zlaqr0.

Array, DIMENSION (*ldz*, *ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., *z* is not referenced.

ldz — INTEGER. The leading dimension of the array *z*.

If *wantz* is .TRUE., then *ldz* ≥ max(1, *ihiz*). Otherwise, *ldz* ≥ 1.

work — REAL for slaqr0

DOUBLE PRECISION for dlaqr0

COMPLEX for claqr0

DOUBLE COMPLEX for zlaqr0.

Workspace array with dimension *lwork*.

lwork — INTEGER. The dimension of the array *work*.

lwork ≥ max(1,*n*) is sufficient, but for the optimal performance a greater workspace may be required, typically as large as 6**n*.

It is recommended to use the workspace query to determine the optimal workspace size. If *lwork*=-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *ilo*, and *ihi*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h

If *info*=0, and *wantt* is .TRUE., then *h* contains the upper quasi-triangular/ triangular matrix *T* from the Schur decomposition (the Schur form).

If *info*=0, and *wantt* is .FALSE., then the contents of *h* are unspecified on exit.

(The output values of *h* when *info* > 0 are given under the description of the *info* parameter below.)

The routine may explicitly set *h*(i,j) for i>j and j=1,2,...*ilo*-1 or j=*ihi*+1, *ihi*+2,...*n*.

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

w

COMPLEX for claqr0

DOUBLE COMPLEX for zlaqr0.

Arrays, DIMENSION(*n*). The computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in *w*(*ilo*:*ihi*). If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *w*(i) = *h*(i,i).

wr, *wi*

REAL for slaqr0

DOUBLE PRECISION for dlaqr0

Arrays, DIMENSION(*ihi*) each. The real and imaginary parts, respectively, of the computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in *wr*(*ilo*:*ihi*) and *wi*(*ilo*:*ihi*). If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with *wi*(*i*)> 0 and *wi*(*i*+1) < 0. If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *wr*(i) = *h*(i,i), and if *h*(i:i+1,i:i+1)is a 2-by-2 diagonal block, then *wi*(i)=sqrt(-*h*(i+1,i)*h(i,i+1)).

z

If *wantz* is .TRUE., then *z(ilo:ihi, iloz:ihiz)* is replaced by *z(ilo:ihi, iloz:ihiz)*U*, where *U* is the orthogonal/unitary Schur factor of *h(ilo:ihi, ilo:ihi)*.

If *wantz* is .FALSE., *z* is not referenced.

(The output values of *z* when *info* > 0 are given under the description of the *info* parameter below.)

info

INTEGER.

= 0: the execution is successful.

> 0: if *info* = *i*, then the routine failed to compute all the eigenvalues. Elements 1:*ilo*-1 and i+1:*n* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

> 0: if *wantt* is .FALSE., then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.

> 0: if *wantt* is .TRUE., then (initial value of *h*)**U* = *U**(final value of *h*, where *U* is an orthogonal/unitary matrix. The final value of *h* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info*+1 through *ihi*.

> 0: if *wantz* is .TRUE., then (final value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*))=(initial value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*)***U*, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).

> 0: if *wantz* is .FALSE., then *z* is not accessed.

?laqr1

Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix H and specified shifts.

Syntax

```
call slaqr1( n, h, ldh, sr1, si1, sr2, si2, v )
call dlaqr1( n, h, ldh, sr1, si1, sr2, si2, v )
call claqr1( n, h, ldh, s1, s2, v )
call zlaqr1( n, h, ldh, s1, s2, v )
```

Include Files

- mkl.fi

Description

Given a 2-by-2 or 3-by-3 matrix *H*, this routine sets *v* to a scalar multiple of the first column of the product *K* = (*H* - *s1***I*)*(*H* - *s2***I*), or *K* = (*H* - (*sr1* + i**si1*)**I*)*(*H* - (*sr2* + i**si2*)**I*) scaling to avoid overflows and most underflows.

It is assumed that either 1) *sr1* = *sr2* and *si1* = -*si2*, or 2) *si1* = *si2* = 0.

This is useful for starting double implicit shift bulges in the QR algorithm.

Input Parameters

n	INTEGER. The order of the matrix *H*. *n* must be equal to 2 or 3.
sr1, *si2*, *sr2*, *si2*	REAL for slaqr1 DOUBLE PRECISION for dlaqr1

Shift values that define *K* in the formula above.

s1, *s2*

COMPLEX for claqr1

DOUBLE COMPLEX for zlaqr1.

Shift values that define *K* in the formula above.

h

REAL for slaqr1

DOUBLE PRECISION for dlaqr1

COMPLEX for claqr1

DOUBLE COMPLEX for zlaqr1.

Array, DIMENSION (*ldh*, *n*), contains 2-by-2 or 3-by-3 matrix *H* in the formula above.

ldh

INTEGER.

The leading dimension of the array *h* just as declared in the calling routine. *ldh* ≥ *n*.

Output Parameters

v

REAL for slaqr1

DOUBLE PRECISION for dlaqr1

COMPLEX for claqr1

DOUBLE COMPLEX for zlaqr1.

Array with dimension (*n*).

A scalar multiple of the first column of the matrix *K* in the formula above.

?laqr2

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call slaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
call dlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
call claqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
call zlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

Include Files

- mkl.fi

Description

The routine accepts as input an upper Hessenberg matrix *H* and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output *H* has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of *H*. It is to be hoped that the final version of *H* has many zero subdiagonal entries.

This subroutine is identical to `?laqr3` except that it avoids recursion by calling `?lahqr` instead of `?laqr4`.

Input Parameters

wantt

`LOGICAL`.

If *wantt* = `.TRUE.`, then the Hessenberg matrix *H* is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine).

If *wantt* = `.FALSE.`, then only enough of *H* is updated to preserve the eigenvalues.

wantz

`LOGICAL`.

If *wantz* = `.TRUE.`, then the orthogonal/unitary matrix *Z* is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine).

If *wantz* = `.FALSE.`, then *Z* is not referenced.

n

`INTEGER`. The order of the Hessenberg matrix *H* and (if *wantz* = `.TRUE.`) the order of the orthogonal/unitary matrix *Z*.

ktop

`INTEGER`.

It is assumed that either *ktop*=1 or *h*(*ktop*,*ktop*-1)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

kbot

`INTEGER`.

It is assumed without a check that either *kbot*=*n* or *h*(*kbot*+1,*kbot*)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

nw

`INTEGER`.

Size of the deflation window. 1 ≤ *nw* ≤ (*kbot*-*ktop*+1).

h

`REAL` for `slaqr2`

`DOUBLE PRECISION` for `dlaqr2`

`COMPLEX` for `claqr2`

`DOUBLE COMPLEX` for `zlaqr2`.

Array, `DIMENSION` (*ldh*, *n*), on input the initial *n*-by-*n* section of *h* stores the Hessenberg matrix *H* undergoing aggressive early deflation.

ldh

`INTEGER`. The leading dimension of the array *h* just as declared in the calling subroutine. *ldh*≥*n*.

iloz, *ihiz* — INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE.. $1 \le iloz \le ihiz \le n$.

z — REAL for slaqr2

DOUBLE PRECISION for dlaqr2

COMPLEX for claqr2

DOUBLE COMPLEX for zlaqr2.

Array, DIMENSION (*ldz*, *n*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., then *z* is not referenced.

ldz — INTEGER. The leading dimension of the array *z* just as declared in the calling subroutine. $ldz \ge 1$.

v — REAL for slaqr2

DOUBLE PRECISION for dlaqr2

COMPLEX for claqr2

DOUBLE COMPLEX for zlaqr2.

Workspace array with dimension (*ldv*, *nw*). An *nw*-by-*nw* work array.

ldv — INTEGER. The leading dimension of the array *v* just as declared in the calling subroutine. $ldv \ge nw$.

nh — INTEGER. The number of column of *t*. $nh \ge nw$.

t — REAL for slaqr2

DOUBLE PRECISION for dlaqr2

COMPLEX for claqr2

DOUBLE COMPLEX for zlaqr2.

Workspace array with dimension (*ldt*, *nw*).

ldt — INTEGER. The leading dimension of the array *t* just as declared in the calling subroutine. $ldt \ge nw$.

nv — INTEGER. The number of rows of work array *wv* available for workspace. $nv \ge nw$.

wv — REAL for slaqr2

DOUBLE PRECISION for dlaqr2

COMPLEX for claqr2

DOUBLE COMPLEX for zlaqr2.

Workspace array with dimension (*ldwv*, *nw*).

ldwv — INTEGER. The leading dimension of the array *wv* just as declared in the calling subroutine. $ldwv \ge nw$.

work — REAL for slaqr2

DOUBLE PRECISION for dlaqr2

COMPLEX for claqr2

DOUBLE COMPLEX for zlaqr2.

Workspace array with dimension *lwork*.

lwork INTEGER. The dimension of the array *work*.

lwork=2*nw) is sufficient, but for the optimal performance a greater workspace may be required.

If *lwork*=-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h On output *h* has been transformed by an orthogonal/unitary similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

work(1) On exit *work*(1) is set to an estimate of the optimal value of *lwork* for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*.

z If *wantz* is .TRUE., then the orthogonal/unitary similarity transformation is accumulated into *z(iloz:ihiz, ilo:ihi)* from the right.

If *wantz* is .FALSE., then *z* is unreferenced.

nd INTEGER. The number of converged eigenvalues uncovered by the routine.

ns INTEGER. The number of unconverged, that is approximate eigenvalues returned in *sr*, *si* or in *sh* that may be used as shifts by the calling subroutine.

sh COMPLEX for claqr2

DOUBLE COMPLEX for zlaqr2.

Arrays, DIMENSION (*kbot*).

The approximate eigenvalues that may be used for shifts are stored in the *sh*(*kbot*-*nd*-*ns*+1) through the *sh*(*kbot*-*nd*).

The converged eigenvalues are stored in the *sh*(*kbot*-*nd*+1) through the *sh*(*kbot*).

sr, *si* REAL for slaqr2

DOUBLE PRECISION for dlaqr2

Arrays, DIMENSION (*kbot*) each.

The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the *sr*(*kbot*-*nd*-*ns*+1) through the *sr*(*kbot*-*nd*), and *si*(*kbot*-*nd*-*ns*+1) through the *si*(*kbot*-*nd*), respectively.

The real and imaginary parts of converged eigenvalues are stored in the *sr*(*kbot*-*nd*+1) through the *sr*(*kbot*), and *si*(*kbot*-*nd*+1) through the *si*(*kbot*), respectively.

?laqr3

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call slaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
call dlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
call claqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
call zlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

Include Files

- mkl.fi

Description

The routine accepts as input an upper Hessenberg matrix *H* and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output *H* has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of *H*. It is to be hoped that the final version of *H* has many zero subdiagonal entries.

Input Parameters

wantt

LOGICAL.

If *wantt* = .TRUE., then the Hessenberg matrix *H* is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine).

If *wantt* = .FALSE., then only enough of *H* is updated to preserve the eigenvalues.

wantz

LOGICAL.

If *wantz* = .TRUE., then the orthogonal/unitary matrix *Z* is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine).

If *wantz* = .FALSE., then *Z* is not referenced.

n

INTEGER. The order of the Hessenberg matrix *H* and (if *wantz* = .TRUE.) the order of the orthogonal/unitary matrix *Z*.

ktop

INTEGER.

It is assumed that either *ktop*=1 or *h*(*ktop*,*ktop-1*)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

kbot INTEGER.

It is assumed without a check that either *kbot*=*n* or *h*(*kbot*+1,*kbot*)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

nw INTEGER.

Size of the deflation window. 1≤*nw*≤(*kbot*-*ktop*+1).

h REAL for slaqr3

DOUBLE PRECISION for dlaqr3

COMPLEX for claqr3

DOUBLE COMPLEX for zlaqr3.

Array, DIMENSION (*ldh*, *n*), on input the initial *n*-by-*n* section of *h* stores the Hessenberg matrix *H* undergoing aggressive early deflation.

ldh INTEGER. The leading dimension of the array *h* just as declared in the calling subroutine. *ldh*≥*n*.

iloz, *ihiz* INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE.. 1≤*iloz*≤*ihiz*≤*n*.

z REAL for slaqr3

DOUBLE PRECISION for dlaqr3

COMPLEX for claqr3

DOUBLE COMPLEX for zlaqr3.

Array, DIMENSION (*ldz*, *n*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., then *z* is not referenced.

ldz INTEGER. The leading dimension of the array *z* just as declared in the calling subroutine. *ldz*≥1.

v REAL for slaqr3

DOUBLE PRECISION for dlaqr3

COMPLEX for claqr3

DOUBLE COMPLEX for zlaqr3.

Workspace array with dimension (*ldv*, *nw*). An *nw*-by-*nw* work array.

ldv INTEGER. The leading dimension of the array *v* just as declared in the calling subroutine. *ldv*≥*nw*.

nh INTEGER. The number of column of *t*. *nh*≥*nw*.

t REAL for slaqr3

DOUBLE PRECISION for dlaqr3

COMPLEX for claqr3

DOUBLE COMPLEX for zlaqr3.

Workspace array with dimension (*ldt*, *nw*).

ldt — INTEGER. The leading dimension of the array *t* just as declared in the calling subroutine. *ldt*≥*nw*.

nv — INTEGER. The number of rows of work array *wv* available for workspace. *nv*≥*nw*.

wv — REAL for slaqr3

DOUBLE PRECISION for dlaqr3

COMPLEX for claqr3

DOUBLE COMPLEX for zlaqr3.

Workspace array with dimension (*ldwv*, *nw*).

ldwv — INTEGER. The leading dimension of the array *wv* just as declared in the calling subroutine. *ldwv*≥*nw*.

work — REAL for slaqr3

DOUBLE PRECISION for dlaqr3

COMPLEX for claqr3

DOUBLE COMPLEX for zlaqr3.

Workspace array with dimension *lwork*.

lwork — INTEGER. The dimension of the array *work*.

lwork=2**nw*) is sufficient, but for the optimal performance a greater workspace may be required.

If *lwork*=-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h — On output *h* has been transformed by an orthogonal/unitary similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

work(1) — On exit *work*(1) is set to an estimate of the optimal value of *lwork* for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*.

z — If *wantz* is .TRUE., then the orthogonal/unitary similarity transformation is accumulated into *z(iloz:ihiz, ilo:ihi)* from the right.

If *wantz* is .FALSE., then *z* is unreferenced.

nd — INTEGER. The number of converged eigenvalues uncovered by the routine.

ns — INTEGER. The number of unconverged, that is approximate eigenvalues returned in *sr*, *si* or in *sh* that may be used as shifts by the calling subroutine.

sh — COMPLEX for claqr3

DOUBLE COMPLEX for zlaqr3.

Arrays, DIMENSION (*kbot*).

The approximate eigenvalues that may be used for shifts are stored in the *sh*(*kbot*-*nd*-*ns*+1) through the *sh*(*kbot*-*nd*).

The converged eigenvalues are stored in the *sh*(*kbot*-*nd*+1) through the *sh*(*kbot*).

sr, *si*

REAL for slaqr3

DOUBLE PRECISION for dlaqr3

Arrays, DIMENSION (*kbot*) each.

The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the *sr*(*kbot*-*nd*-*ns*+1) through the *sr*(*kbot*-*nd*), and *si*(*kbot*-*nd*-*ns*+1) through the *si*(*kbot*-*nd*), respectively.

The real and imaginary parts of converged eigenvalues are stored in the *sr*(*kbot*-*nd*+1) through the *sr*(*kbot*), and *si*(*kbot*-*nd*+1) through the *si*(*kbot*), respectively.

?laqr4

Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.

Syntax

call slaqr4(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *h*, *ldh*, *wr*, *wi*, *iloz*, *ihiz*, *z*, *ldz*, *work*, *lwork*, *info*)

call dlaqr4(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *h*, *ldh*, *wr*, *wi*, *iloz*, *ihiz*, *z*, *ldz*, *work*, *lwork*, *info*)

call claqr4(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *h*, *ldh*, *w*, *iloz*, *ihiz*, *z*, *ldz*, *work*, *lwork*, *info*)

call zlaqr4(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *h*, *ldh*, *w*, *iloz*, *ihiz*, *z*, *ldz*, *work*, *lwork*, *info*)

Include Files

- mkl.fi

Description

The routine computes the eigenvalues of a Hessenberg matrix *H*, and, optionally, the matrices *T* and *Z* from the Schur decomposition $H=Z*T*Z^H$, where *T* is an upper quasi-triangular/triangular matrix (the Schur form), and *Z* is the orthogonal/unitary matrix of Schur vectors.

Optionally *Z* may be postmultiplied into an input orthogonal/unitary matrix *Q* so that this routine can give the Schur factorization of a matrix *A* which has been reduced to the Hessenberg form *H* by the orthogonal/unitary matrix *Q*: $A = Q*H*Q^H = (QZ)*H*(QZ)^H$.

This routine implements one level of recursion for ?laqr0. It is a complete implementation of the small bulge multi-shift QR algorithm. It may be called by ?laqr0 and, for large enough deflation window size, it may be called by ?laqr3. This routine is identical to ?laqr0 except that it calls ?laqr2 instead of ?laqr3.

Input Parameters

wantt

LOGICAL.

If *wantt* = .TRUE., the full Schur form T is required;

If *wantt* = .FALSE., only eigenvalues are required.

wantz

LOGICAL.

If *wantz* = .TRUE., the matrix of Schur vectors *Z* is required;

If *wantz* = .FALSE., Schur vectors are not required.

n

INTEGER. The order of the Hessenberg matrix *H*. ($n \geq 0$).

ilo, *ihi*

INTEGER.

It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*, and if $ilo > 1$ then $h(ilo, ilo-1) = 0$.

ilo and *ihi* are normally set by a previous call to cgebal, and then passed to cgehrd when the matrix output by cgebal is reduced to Hessenberg form. Otherwise, *ilo* and *ihi* should be set to 1 and *n*, respectively.

If $n > 0$, then $1 \leq ilo \leq ihi \leq n$.

If $n=0$, then $ilo=1$ and $ihi=0$

h

REAL for slaqr4

DOUBLE PRECISION for dlaqr4

COMPLEX for claqr4

DOUBLE COMPLEX for zlaqr4.

Array, DIMENSION (*ldh*, *n*), contains the upper Hessenberg matrix *H*.

ldh

INTEGER. The leading dimension of the array *h*. $ldh \geq \max(1, n)$.

iloz, *ihiz*

INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE., $1 \leq iloz \leq ilo$; $ihi \leq ihiz \leq n$.

z

REAL for slaqr4

DOUBLE PRECISION for dlaqr4

COMPLEX for claqr4

DOUBLE COMPLEX for zlaqr4.

Array, DIMENSION (*ldz*, *ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., *z* is not referenced.

ldz

INTEGER. The leading dimension of the array *z*.

If *wantz* is .TRUE., then $ldz \geq \max(1, ihiz)$. Otherwise, $ldz \geq 1$.

work

REAL for slaqr4

DOUBLE PRECISION for dlaqr4

COMPLEX for claqr4

DOUBLE COMPLEX for zlaqr4.

Workspace array with dimension *lwork*.

lwork INTEGER. The dimension of the array *work*.

lwork ≥ max(1,*n*) is sufficient, but for the optimal performance a greater workspace may be required, typically as large as 6**n*.

It is recommended to use the workspace query to determine the optimal workspace size. If *lwork*=-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *ilo*, and *ihi*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h If *info*=0, and *wantt* is .TRUE., then *h* contains the upper quasi-triangular/triangular matrix *T* from the Schur decomposition (the Schur form).

If *info*=0, and *wantt* is .FALSE., then the contents of *h* are unspecified on exit.

(The output values of *h* when *info* > 0 are given under the description of the *info* parameter below.)

The routines may explicitly set *h*(i,j) for i>j and j=1,2,...*ilo*-1 or j=*ihi*+1, *ihi*+2,...*n*.

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

w COMPLEX for claqr4

DOUBLE COMPLEX for zlaqr4.

Arrays, DIMENSION(*n*). The computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in *w*(*ilo*:*ihi*). If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *w*(i) = *h*(i,i).

wr, *wi* REAL for slaqr4

DOUBLE PRECISION for dlaqr4

Arrays, DIMENSION(*ihi*) each. The real and imaginary parts, respectively, of the computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in the *wr*(*ilo*:*ihi*) and *wi*(*ilo*:*ihi*). If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with *wi*(*i*)> 0 and *wi*(*i*+1) < 0. If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *wr*(i) = *h*(i,i), and if *h*(i:i+1,i:i+1) is a 2-by-2 diagonal block, then *wi*(i)=sqrt(-*h*(i+1,i)**h*(i,i+1)).

z If *wantz* is .TRUE., then *z(ilo:ihi, iloz:ihiz)* is replaced by *z(ilo:ihi, iloz:ihiz)***U*, where *U* is the orthogonal/unitary Schur factor of *h(ilo:ihi, ilo:ihi)*.

If *wantz* is .FALSE., *z* is not referenced.

(The output values of *z* when *info* > 0 are given under the description of the *info* parameter below.)

info INTEGER.

= 0: the execution is successful.

> 0: if *info* = *i*, then the routine failed to compute all the eigenvalues. Elements 1:*ilo*-1 and i+1:*n* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

> 0: if *wantt* is .FALSE., then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.

> 0: if *wantt* is .TRUE., then (initial value of *h*)**U* = *U**(final value of *h*, where *U* is an orthogonal/unitary matrix. The final value of *h* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info*+1 through *ihi*.

> 0: if *wantz* is .TRUE., then (final value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*))=(initial value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*)***U*, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).

> 0: if *wantz* is .FALSE., then *z* is not accessed.

?laqr5

Performs a single small-bulge multi-shift QR sweep.

Syntax

```
call slaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz, ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
call dlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz, ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
call claqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz, ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
call zlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz, ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

Include Files

- mkl.fi

Description

This auxiliary routine called by ?laqr0 performs a single small-bulge multi-shift QR sweep.

Input Parameters

wantt LOGICAL.

wantt = .TRUE. if the quasi-triangular/triangular Schur factor is computed.

wantt is set to .FALSE. otherwise.

wantz

LOGICAL.

wantz = .TRUE. if the orthogonal/unitary Schur factor is computed.

wantz is set to .FALSE. otherwise.

kacc22

INTEGER. Possible values are 0, 1, or 2.

Specifies the computation mode of far-from-diagonal orthogonal updates.

= 0: the routine does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries.

= 1: the routine accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.

= 2: the routine accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.

n

INTEGER. The order of the Hessenberg matrix *H* upon which the routine operates.

ktop, *kbot*

INTEGER.

It is assumed without a check that either *ktop*=1 or *h*(*ktop*,*ktop-1*)=0, and either *kbot*=*n* or *h*(*kbot*+1,*kbot*)=0.

nshfts

INTEGER.

Number of simultaneous shifts, must be positive and even.

sr, *si*

REAL for slaqr5

DOUBLE PRECISION for dlaqr5

Arrays, DIMENSION (*nshfts*) each.

sr contains the real parts and *si* contains the imaginary parts of the *nshfts* shifts of origin that define the multi-shift QR sweep.

s

COMPLEX for claqr5

DOUBLE COMPLEX for zlaqr5.

Arrays, DIMENSION (*nshfts*).

s contains the shifts of origin that define the multi-shift QR sweep.

h

REAL for slaqr5

DOUBLE PRECISION for dlaqr5

COMPLEX for claqr5

DOUBLE COMPLEX for zlaqr5.

Array, DIMENSION (*ldh*, *n*), on input contains the Hessenberg matrix.

ldh — INTEGER. The leading dimension of the array *h* just as declared in the calling routine. *ldh* ≥ max(1, *n*).

iloz, *ihiz* — INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE.. 1 ≤ *iloz* ≤ *ihiz* ≤ *n*.

z — REAL for slaqr5

DOUBLE PRECISION for dlaqr5

COMPLEX for claqr5

DOUBLE COMPLEX for zlaqr5.

Array, DIMENSION (*ldz*, *ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., then *z* is not referenced.

ldz — INTEGER. The leading dimension of the array *z* just as declared in the calling routine. *ldz* ≥ *n*.

v — REAL for slaqr5

DOUBLE PRECISION for dlaqr5

COMPLEX for claqr5

DOUBLE COMPLEX for zlaqr5.

Workspace array with dimension (*ldv*, *nshfts*/2).

ldv — INTEGER. The leading dimension of the array *v* just as declared in the calling routine. *ldv* ≥ 3.

u — REAL for slaqr5

DOUBLE PRECISION for dlaqr5

COMPLEX for claqr5

DOUBLE COMPLEX for zlaqr5.

Workspace array with dimension (*ldu*, 3**nshfts*-3).

ldu — INTEGER. The leading dimension of the array *u* just as declared in the calling routine. *ldu* ≥ 3**nshfts*-3.

nh — INTEGER. The number of column in the array *wh* available for workspace. *nh* ≥ 1.

wh — REAL for slaqr5

DOUBLE PRECISION for dlaqr5

COMPLEX for claqr5

DOUBLE COMPLEX for zlaqr5.

Workspace array with dimension (*ldwh*, *nh*)

ldwh — INTEGER. The leading dimension of the array *wh* just as declared in the calling routine. *ldwh* ≥ 3**nshfts*-3

nv — INTEGER. The number of rows of the array *wv* available for workspace. *nv* ≥ 1.

wv
: REAL for slaqr5
: DOUBLE PRECISION for dlaqr5
: COMPLEX for claqr5
: DOUBLE COMPLEX for zlaqr5.
: Workspace array with dimension (*ldwv*, 3**nshfts*-3).

ldwv
: INTEGER. The leading dimension of the array *wv* just as declared in the calling routine. *ldwv* ≥ *nv*.

Output Parameters

sr, *si*
: On output, may be reordered.

h
: On output a multi-shift QR Sweep with shifts *sr*(j)+i**si*(j) or *s*(j) is applied to the isolated diagonal block in rows and columns *ktop* through *kbot* .

z
: If *wantz* is .TRUE., then the QR Sweep orthogonal/unitary similarity transformation is accumulated into *z(iloz:ihiz, ilo:ihi)* from the right.
: If *wantz* is .FALSE., then *z* is unreferenced.

?laqsb

Scales a symmetric band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call slaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call dlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call claqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call zlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a symmetric band matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo
: CHARACTER*1.
: Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored.
: If *uplo* = 'U': upper triangular.
: If *uplo* = 'L': lower triangular.

n
: INTEGER. The order of the matrix *A*.

n ≥ 0.

kd — INTEGER. The number of super-diagonals of the matrix *A* if *uplo* = 'U', or the number of sub-diagonals if *uplo* = 'L'.

kd ≥ 0.

ab — REAL for slaqsb

DOUBLE PRECISION for dlaqsb

COMPLEX for claqsb

DOUBLE COMPLEX for zlaqsb

Array, DIMENSION (*ldab*,*n*). On entry, the upper or lower triangle of the symmetric band matrix *A*, stored in the first *kd*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

if *uplo* = 'U', *ab*(*kd*+1+*i*-*j*,*j*) = A(*i*,*j*) for max(1,*j*-*kd*) ≤ *i* ≤ *j*;

if *uplo* = 'L', *ab*(1+*i*-*j*,*j*) = A(*i*,*j*) for *j* ≤ *i* ≤ min(*n*,*j*+*kd*).

ldab — INTEGER. The leading dimension of the array *ab*.

ldab ≥ *kd*+1.

s — REAL for slaqsb/claqsb

DOUBLE PRECISION for dlaqsb/zlaqsb

Array, DIMENSION (*n*). The scale factors for A.

scond — REAL for slaqsb/claqsb

DOUBLE PRECISION for dlaqsb/zlaqsb

Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax — REAL for slaqsb/claqsb

DOUBLE PRECISION for dlaqsb/zlaqsb

Absolute value of largest matrix entry.

Output Parameters

ab — On exit, if *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization of the band matrix *A* that can be $A = U^T*U$ or $A = L*L^T$ for real flavors and $A = U^H*U$ or $A = L*L^H$ for complex flavors, in the same storage format as *A*.

equed — CHARACTER*1.

Specifies whether or not equilibration was done.

If *equed* = 'N': No equilibration.

If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(*s*)*A*diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqsp

Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.

Syntax

```
call slaqsp( uplo, n, ap, s, scond, amax, equed )
call dlaqsp( uplo, n, ap, s, scond, amax, equed )
call claqsp( uplo, n, ap, s, scond, amax, equed )
call zlaqsp( uplo, n, ap, s, scond, amax, equed )
```

Include Files

- mkl.fi

Description

The routine ?laqsp equilibrates a symmetric matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored.

If *uplo* = 'U': upper triangular.

If *uplo* = 'L': lower triangular.

n

INTEGER. The order of the matrix *A*. $n \geq 0$.

ap

REAL for slaqsp

DOUBLE PRECISION for dlaqsp

COMPLEX for claqsp

DOUBLE COMPLEX for zlaqsp

Array, DIMENSION ($n(n+1)/2$).

On entry, the upper or lower triangle of the symmetric matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;

if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

s

REAL for slaqsp/claqsp

DOUBLE PRECISION for dlaqsp/zlaqsp

Array, DIMENSION (*n*). The scale factors for A.

scond

REAL for slaqsp/claqsp

DOUBLE PRECISION for dlaqsp/zlaqsp

Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax

REAL for slaqsp/claqsp

DOUBLE PRECISION for dlaqsp/zlaqsp

Absolute value of largest matrix entry.

Output Parameters

ap

On exit, the equilibrated matrix: diag(*s*)*A*diag(*s*), in the same storage format as *A*.

equed

CHARACTER*1.

Specifies whether or not equilibration was done.

If *equed* = 'N': No equilibration.

If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(*s*)*A*diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by ?syequ*,* ?syequb*,* ?poequ *,* *or* ?poequb*.*

Syntax

```
call slaqsy( uplo, n, a, lda, s, scond, amax, equed )
call dlaqsy( uplo, n, a, lda, s, scond, amax, equed )
call claqsy( uplo, n, a, lda, s, scond, amax, equed )
call zlaqsy( uplo, n, a, lda, s, scond, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a symmetric matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored.

If *uplo* = 'U': upper triangular.

If *uplo* = 'L': lower triangular.

n

INTEGER. The order of the matrix *A*.

n ≥ 0.

a

REAL for slaqsy

DOUBLE PRECISION for dlaqsy

COMPLEX for claqsy

DOUBLE COMPLEX for zlaqsy

Array, DIMENSION (*lda*,*n*). On entry, the symmetric matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda

INTEGER. The leading dimension of the array *a*.

lda ≥ max(*n*,1).

s

REAL for slaqsy/claqsy

DOUBLE PRECISION for dlaqsy/zlaqsy

Array, DIMENSION (*n*). The scale factors for A.

scond

REAL for slaqsy/claqsy

DOUBLE PRECISION for dlaqsy/zlaqsy

Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax

REAL for slaqsy/claqsy

DOUBLE PRECISION for dlaqsy/zlaqsy

Absolute value of largest matrix entry.

Output Parameters

a

On exit, if *equed* = 'Y', the equilibrated matrix: diag(*s*)*A*diag(*s*).

equed

CHARACTER*1.

Specifies whether or not equilibration was done.

If *equed* = 'N': No equilibration.

If *equed* = 'Y': Equilibration was done, i.e., *A* has been replaced by diag(*s*)*A*diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqtr

Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.

Syntax

```
call slaqtr( ltran, lreal, n, t, ldt, b, w, scale, x, work, info )
call dlaqtr( ltran, lreal, n, t, ldt, b, w, scale, x, work, info )
```

Include Files

- mkl.fi

Description

The routine ?laqtr solves the real quasi-triangular system

op(*T*) * *p* = *scale***c*, if *lreal* = .TRUE.

or the complex quasi-triangular systems

op(*T* + *iB*)*(*p*+*iq*) = *scale**(*c*+*id*), if *lreal* = .FALSE.

in real arithmetic, where *T* is upper quasi-triangular.

If *lreal* = .FALSE., then the first diagonal block of *T* must be 1-by-1, *B* is the specially structured matrix

$$B = \begin{bmatrix} b_1 & b_2 & \dots & \dots & b_n \\ & W & & & \\ & & W & & \\ & & & \dots & \\ & & & & W \end{bmatrix}$$

`op(A) = A` or A^T, A^T denotes the transpose of matrix A.

On input,

$$x = \begin{bmatrix} c \\ d \end{bmatrix}, \text{ on output } x = \begin{bmatrix} p \\ q \end{bmatrix}$$

This routine is designed for the condition number estimation in routine `?trsna`.

Input Parameters

ltran — `LOGICAL`.

On entry, *ltran* specifies the option of conjugate transpose:

= `.FALSE.`, `op(T + iB) = T + iB`,

= `.TRUE.`, `op(T + iB) = (T + iB)`^{T}.

lreal — `LOGICAL`.

On entry, *lreal* specifies the input matrix structure:

= `.FALSE.`, the input is complex

= `.TRUE.`, the input is real.

n — `INTEGER`.

On entry, *n* specifies the order of `T + iB`. *n*≥ `0`.

t — `REAL` for `slaqtr`

DOUBLE PRECISION for dlaqtr

Array, dimension (*ldt*,*n*). On entry, *t* contains a matrix in Schur canonical form. If *lreal* = .FALSE., then the first diagonal block of *t* must be 1-by-1.

ldt INTEGER. The leading dimension of the matrix *T*.

ldt≥ max(1,*n*).

b REAL for slaqtr

DOUBLE PRECISION for dlaqtr

Array, dimension (*n*). On entry, *b* contains the elements to form the matrix *B* as described above. If *lreal* = .TRUE., *b* is not referenced.

w REAL for slaqtr

DOUBLE PRECISION for dlaqtr

On entry, *w* is the diagonal element of the matrix *B*.

If *lreal* = .TRUE., *w* is not referenced.

x REAL for slaqtr

DOUBLE PRECISION for dlaqtr

Array, dimension (2*n*). On entry, *x* contains the right hand side of the system.

work REAL for slaqtr

DOUBLE PRECISION for dlaqtr

Workspace array, dimension (*n*).

Output Parameters

scale REAL for slaqtr

DOUBLE PRECISION for dlaqtr

On exit, *scale* is the scale factor.

x On exit, *X* is overwritten by the solution.

info INTEGER.

If *info* = 0: successful exit.

If *info* = 1: the some diagonal 1-by-1 block has been perturbed by a small number smin to keep nonsingularity.

If *info* = 2: the some diagonal 2-by-2 block has been perturbed by a small number in ?laln2 to keep nonsingularity.

NOTE
For higher speed, this routine does not check the inputs for errors.

?laqz0

Implements the multishift QZ method with aggressive early deflation for finding the generalized eigenvalues of the matrix pair (A,B).

Syntax

FORTRAN 77:

```
call slaqz0(wants, wantq, wantz, n, ilo, ihi, a, lda, b, ldb, alphar, alphai, beta, q, ldq, z, ldz, work, lwork, rec, info)
call dlaqz0(wants, wantq, wantz, n, ilo, ihi, a, lda, b, ldb, alphar, alphai, beta, q, ldq, z, ldz, work, lwork, rec, info)
call claqz0(wants, wantq, wantz, n, ilo, ihi, a, lda, b, ldb, alpha, beta, q, ldq, z, ldz, work, lwork, rwork, rec, info)
call zlaqz0(wants, wantq, wantz, n, ilo, ihi, a, lda, b, ldb, alpha, beta, q, ldq, z, ldz, work, lwork, rwork, rec, info)
```

Include Files

The FORTRAN 77 interfaces are specified in the `mkl_lapack.fi` and `mkl_lapack.h` include files.

Description

The routine computes the eigenvalues of a real/complex matrix pair (A,B), where A is an upper Hessenberg matrix and B is upper triangular, using the double-shift version (for real flavors) or single-shift version (for complex flavors) of the multishift QZ method with aggressive early deflation.

For real flavors:

If `wants` = 'S', then the Hessenberg-triangular pair (A,B) is also reduced to generalized Schur form:

```
A = Q*S*Z^T, B = Q *P*Z^T
```

where Q and Z are orthogonal matrices, P is an upper triangular matrix, and S is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks. The 1-by-1 blocks correspond to real eigenvalues of the matrix pair (A,B) and the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues.

Additionally, the 2-by-2 upper triangular diagonal blocks of P corresponding to 2-by-2 blocks of S are reduced to positive diagonal form; that is, if `S(j+1,j)` is non-zero, then `P(j+1,j) = P(j,j+1) = 0`, `P(j,j) > 0`, and `P(j+1,j+1) > 0`.

For complex flavors:

If `wants` = 'S', then the Hessenberg-triangular pair (A,B) is also reduced to generalized Schur form:

```
A = Q* S*Z^H, B = Q*P*Z^H
```

where Q and Z are unitary matrices, P is an upper triangular matrix, and S is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks.

For all function flavors:

Optionally, the orthogonal/unitary matrix Q from the generalized Schur factorization may be postmultiplied into an input matrix Q1, and the orthogonal/unitary matrix Z may be postmultiplied into an input matrix Z1.

If Q1 and Z1 are the orthogonal/unitary matrices from `?gghrd` that reduced the matrix pair (A1,B1) to generalized upper Hessenberg form (A,B), then the output matrices Q1*Q and Z1*Z are the orthogonal/unitary factors from the generalized Schur factorization of (A,B):

A1 = (Q1*Q)**S** ***(Z1*Z)^H, B1 = (Q1*Q)*P**(Z1*Z)^H

To avoid overflow, eigenvalues of the matrix pair (A,B) are computed as a pair of values (alpha,beta). For `claqz0/zlaqz0`, alpha and beta are complex, and for `slaqz0/dlaqz0`, alpha is complex and beta real.

If beta is nonzero, λ = alpha/beta is an eigenvalue of the generalized nonsymmetric eigenvalue problem (GNEP):

```
A*x = λ*B*x
```

and if alpha is nonzero, μ = beta/alpha is an eigenvalue of the alternate form of the GNEP:

```
μ*A*y = B*y
```

Real eigenvalues (for real flavors) or the values of alpha and beta for the i-th eigenvalue (for complex flavors) can be read directly from the generalized Schur form:

```
alpha = S(i,i), beta = P(i,i)
```

Input Parameters

The data types are given for the Fortran interface. A *datatype* placeholder, if present, is used for the C interface data types.

Refer to the *C Interface Conventions* topic in the C-based reference for the C interface principal conventions and type definitions.

wants

`CHARACTER*1`. Specifies the operations to be performed. Must be 'E' or 'S'.

If wants = 'E', then compute eigenvalues only.

If wants = 'S', then compute eigenvalues and the Schur form.

wantq

`CHARACTER*1`. Must be 'N', 'I', or 'V'.

If wantq = 'N', left Schur vectors `(q)` are not computed.

If wantq = 'I', `q` is initialized to the unit matrix and the matrix of left Schur vectors of `(A, B)` is returned.

If wantq = 'V', q must contain an orthogonal/unitary matrix Q1 on entry and the product `Q1*Q` is returned.

wantz

`CHARACTER*1`. Must be 'N', 'I', or 'V'.

If wantz = 'N', right Schur vectors `(z)` are not computed.

If wantz = 'I', `z` is initialized to the unit matrix and the matrix of right Schur vectors of `(A,B)` is returned.

If wantz = 'V', `z` must contain an orthogonal/unitary matrix `Z1` on entry and the product `Z1*Z` is returned.

n

`INTEGER`. The order of the matrices A, B, Q, and Z (`n ≥ 0`).

ilo, ihi

`INTEGER`. `ilo` and `ihi` mark the rows and columns of A which are in Hessenberg form. It is assumed that A is already upper triangular in rows and columns `1:ilo-1` and `ihi+1:n`.

Constraint:

If `n > 0`, then `1 ≤ ilo ≤ ihi ≤ n`.

If `n = 0`, then `ilo = 1` and `ihi = 0`.

a, b, q, z, work

`REAL` for `slaqz0`..

`DOUBLE PRECISION` for `dlaqz0`.

`COMPLEX` for `claqz0`.

`DOUBLE COMPLEX` for `zlaqz0`.

Arrays:

On entry, `a(lda,*)` contains the n-by-n upper Hessenberg matrix A. The second dimension of `a` must be at least `max(1, n)`.

On entry, `b(ldb,*)` contains the n-by-n upper triangular matrix B. The second dimension of `b` must be at least `max(1, n)`.

q (ldq,*):

On entry, if `wantq` = 'V', this array contains the orthogonal/unitary matrix Q1 used in the reduction of (A1,B1) to generalized Hessenberg form.

If `wantq` = 'N', then `q` is not referenced. The second dimension of `q` must be at least `max(1, n)`.

z (ldz,*):

On entry, if `wantz` = 'V', this array contains the orthogonal/unitary matrix Z1 used in the reduction of (A1,B1) to generalized Hessenberg form (A, B).

If `wantz` = 'N', then `z` is not referenced.

The second dimension of `z` must be at least `max(1, n)`.

`work` is a workspace array; its dimension is `max(1, lwork)`.

lda — `INTEGER`. The first dimension of `a`; at least `max(1, n)`.

ldb — `INTEGER`. The first dimension of `b`; at least `max(1, n)`.

ldq — `INTEGER`. The first dimension of `q`.

If `wantq` = 'N', then `ldq ≥ 1`.

If `wantq` = 'I' or 'V', then `ldq ≥ max(1, n)`.

ldz — `INTEGER`. The first dimension of `z`.

If `wantq` = 'N', then `ldz ≥ 1`.

If `wantq` = 'I'or 'V', then `ldz ≥ max(1, n)`.

rec — `INTEGER`. Indicates the current recursion level. Should be set to 0 on first call.

lwork — `INTEGER`. The dimension of the array `work`. `lwork ≥ max(1, n)`.

If `lwork = -1`, then a workspace query is assumed; the routine calculates only the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by xerbla.

For details, see Application Notes.

rwork — `REAL` for `claqz0`.

`DOUBLE PRECISION` for `zlaqz0`.

Workspace array with DIMENSION at least `max(1, n)`. Used in complex flavors only.

Output Parameters

a

For real flavors:

If `wants` = 'S', then on exit `a` contains the upper quasi-triangular matrix S from the generalized Schur factorization; 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with `a(i,i) = a(i+1, i+1)` and `a(i+1, i) * a(i, i+1) < 0`.

If `wants` = 'E', then on exit the diagonal blocks of a match those of S, but the rest of a is unspecified.

For complex flavors:

If `wants` = 'S', then on exit `a` contains the upper triangular matrix S from the generalized Schur factorization.

If `wants` = 'E', then on exit the diagonal of a matches that of S, but the rest of `a` is unspecified.

b

If `wants` = 'S', then on exit `b` contains the upper triangular matrix P from the generalized Schur factorization.

For real flavors:

2-by-2 diagonal blocks of P corresponding to 2-by-2 blocks of S are reduced to positive diagonal form; that is, if `b(j+1,j)` is non-zero, then `b(j+1,j)=b(j,j+1)=0` and `b(j,j)` and `b(j+1,j+1)` will be positive.

If `wants` = 'E', then on exit the diagonal blocks of `b` match those of P, but the rest of `b` is unspecified.

For complex flavors:

if wants = 'E', then on exit the diagonal of `b` matches that of P, but the rest of `b` is unspecified.

alphar, alphai

`REAL` for slaqz0;

`DOUBLE PRECISION` for dlaqz0.

Arrays, DIMENSION at least `max(1, n)`. The real and imaginary parts, respectively, of each scalar `alpha` define an eigenvalue of GNEP.

If `alphai(j)` is zero, then the `j`-th eigenvalue is real; if positive, then the `j`-th and `(j+1)`-th eigenvalues are a complex conjugate pair, with `alphai(j+1) = -alphai(j)`.

alpha

`COMPLEX` for `claqz0`.

`DOUBLE COMPLEX` for `zlaqz0`.

Array, DIMENSION at least `max(1, n)`.

The complex scalars `alpha` defines the eigenvalues of GNEP. `alphai(i) = S(i,i)` in the generalized Schur factorization.

beta

`REAL` for slaqz0.

`DOUBLE PRECISION` for dlaqz0

`COMPLEX` for claqz0

`DOUBLE COMPLEX` for zlaqz0.

Array, DIMENSION at least `max(1, n)`.

For real flavors:

The scalars `beta` defines the eigenvalues of GNEP.

Together, the quantities `alpha = (alphar(j), alphai(j))` and `beta = beta(j)` represents the `j`-th eigenvalue of the matrix pair (A,B), in one of the forms `lambda = alpha/beta` or `mu = beta/alpha`. In general, because either lambda or mu may overflow, they should not be computed.

For complex flavors:

The real non-negative scalars `beta` defines the eigenvalues of GNEP. Together, the quantities `alpha = alpha(j)` and `beta = beta(j)` represent the `j`-th eigenvalue of the matrix pair (A,B), in one of the forms `lambda = alpha/beta` or `mu = beta/alpha`. In general, because either lambda or mu may overflow, they should not be computed.

q

On exit, if `wantq` = 'I', `q` is overwritten by the orthogonal/unitary matrix of left Schur vectors of the pair (A,B), and if `wantq` = 'V', `q` is overwritten by the orthogonal/unitary matrix of left Schur vectors of (A1, B1).

z

On exit, if `wantz` = 'I', `z` is overwritten by the orthogonal/unitary matrix of right Schur vectors of the pair (A, B), and if `wantz` = 'V', `z` is overwritten by the orthogonal/unitary matrix of right Schur vectors of (A1, B1).

`work(1)`

If `info` ≥ `0`, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

info

`INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the `i`-th parameter had an illegal value.

If `info = 1,..., n`, the QZ iteration did not converge.

(A,B) is not in Schur form, but `alphar(i)`, `alphai(i)` (for real flavors), `alpha(i)` (for complex flavors), and `beta(i), i=info+1,..., n` should be correct.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of the admissible `lwork` sizes no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value of `work(1)` for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array `work`. This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?lar1v

Computes the (scaled) r-th column of the inverse of the submatrix in rows b1 through bn of tridiagonal matrix.

Syntax

```
call slar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )
call dlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )
call clar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )
call zlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )
```

Include Files

- `mkl.fi`

Description

The routine `?lar1v` computes the (scaled) *r*-th column of the inverse of the submatrix in rows *b1* through *bn* of the tridiagonal matrix $L*D*L^T - \lambda*I$. When λ is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, *r* corresponds to the index where the eigenvector is largest in magnitude.

The following steps accomplish this computation :

- Stationary *qd* transform, $L*D*L^T - \lambda*I = L(+)*D(+)*L(+)^T$
- Progressive *qd* transform, $L*D*L^T - \lambda*I = U(-)*D(-)*U(-)^T$,
- Computation of the diagonal elements of the inverse of $L*D*L^T - \lambda*I$ by combining the above transforms, and choosing *r* as the index where the diagonal of the inverse is (one of the) largest in magnitude.
- Computation of the (scaled) *r*-th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

n
: `INTEGER`. The order of the matrix $L*D*L^T$.

b1
: `INTEGER`. First index of the submatrix of $L*D*L^T$.

bn
: `INTEGER`. Last index of the submatrix of $L*D*L^T$.

lambda
: `REAL` for `slar1v`/`clar1v`

`DOUBLE PRECISION` for `dlar1v`/`zlar1v`

The shift. To compute an accurate eigenvector, *lambda* should be a good approximation to an eigenvalue of $L*D*L^T$.

l
: `REAL` for `slar1v`/`clar1v`

DOUBLE PRECISION for dlar1v/zlar1v

Array, DIMENSION (*n*-1).

The (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L*, in elements 1 to *n*-1.

d

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Array, DIMENSION (*n*).

The *n* diagonal elements of the diagonal matrix *D*.

ld

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Array, DIMENSION (*n*-1).

The *n*-1 elements L_i*D_i.

lld

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Array, DIMENSION (*n*-1).

The *n*-1 elements $L_i*L_i*D_i$.

pivmin

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

The minimum pivot in the Sturm sequence.

gaptol

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.

z

REAL for slar1v

DOUBLE PRECISION for dlar1v

COMPLEX for clar1v

DOUBLE COMPLEX for zlar1v

Array, DIMENSION (*n*). All entries of *z* must be set to 0.

wantnc

LOGICAL.

Specifies whether *negcnt* has to be computed.

r

INTEGER.

The twist index for the twisted factorization used to compute *z*. On input, $0 \le r \le n$. If *r* is input as 0, *r* is set to the index where $(L*D*L^T - lambda*I)^{-1}$ is largest in magnitude. If $1 \le r \le n$, *r* is unchanged.

work

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Workspace array, DIMENSION (4*n).

Output Parameters

z

REAL for slar1v

DOUBLE PRECISION for dlar1v

COMPLEX for clar1v

DOUBLE COMPLEX for zlar1v

Array, DIMENSION (*n*). The (scaled) *r*-th column of the inverse. *z*(*r*) is returned to be 1.

negcnt

INTEGER. If *wantnc* is .TRUE. then *negcnt* = the number of pivots < *pivmin* in the matrix factorization $L*D*L^T$, and *negcnt* = -1 otherwise.

ztz

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

The square of the 2-norm of *z*.

mingma

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

The reciprocal of the largest (in magnitude) diagonal element of the inverse of $L*D*L^T$ - *lambda*I*.

r

On output, *r* is the twist index used to compute *z*. Ideally, *r* designates the position of the maximum entry in the eigenvector.

isuppz

INTEGER. Array, DIMENSION (2). The support of the vector in *Z*, that is, the vector *z* is nonzero only in elements *isuppz*(1) through *isuppz*(2).

nrminv

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Equals 1/sqrt(*ztz*).

resid

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

The residual of the FP vector.

resid = ABS(*mingma*)/sqrt(*ztz*).

rqcorr

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

The Rayleigh Quotient correction to *lambda*.

rqcorr = *mingma*/*ztz*.

?lar2v

Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.

Syntax

```
call slar2v( n, x, y, z, incx, c, s, incc )
call dlar2v( n, x, y, z, incx, c, s, incc )
call clar2v( n, x, y, z, incx, c, s, incc )
call zlar2v( n, x, y, z, incx, c, s, incc )
```

Include Files

- mkl.fi

Description

The routine ?lar2v applies a vector of real/complex plane rotations with real cosines from both sides to a sequence of 2-by-2 real symmetric or complex Hermitian matrices, defined by the elements of the vectors x, y and z. For i = 1,2,...,*n*

$$\begin{bmatrix} x_i & z_i \\ \mathrm{conjg}(z_i) & y_i \end{bmatrix} := \begin{bmatrix} c(i) & \mathrm{conjg}(s(i)) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i & z_i \\ \mathrm{conjg}(z_i) & y_i \end{bmatrix} \begin{bmatrix} c(i) & -\mathrm{conjg}(s(i)) \\ s(i) & c(i) \end{bmatrix}$$

Input Parameters

n	INTEGER. The number of plane rotations to be applied.
x, *y*, *z*	REAL for slar2v DOUBLE PRECISION for dlar2v COMPLEX for clar2v DOUBLE COMPLEX for zlar2v Arrays, DIMENSION (1+(*n*-1)**incx*) each. Contain the vectors x, y and z, respectively. For all flavors of ?lar2v, elements of *x* and *y* are assumed to be real.
incx	INTEGER. The increment between elements of *x*, *y*, and *z*. *incx* > 0.
c	REAL for slar2v/clar2v DOUBLE PRECISION for dlar2v/zlar2v Array, DIMENSION (1+(*n*-1)**incc*). The cosines of the plane rotations.
s	REAL for slar2v DOUBLE PRECISION for dlar2v COMPLEX for clar2v DOUBLE COMPLEX for zlar2v Array, DIMENSION (1+(*n*-1)**incc*). The sines of the plane rotations.
incc	INTEGER. The increment between elements of *c* and *s*. *incc* > 0.

Output Parameters

x, *y*, *z*	Vectors *x*, *y* and *z*, containing the results of transform.

?laran

Returns a random real number from a uniform distribution.

Syntax

```
res = slaran (iseed)
res = dlaran (iseed)
```

Description

The `?laran` routine returns a random real number from a uniform (0,1) distribution. This routine uses a multiplicative congruential method with modulus 2^{48} and multiplier 33952834046453. 48-bit integers are stored in four integer array elements with 12 bits per element. Hence the routine is portable across machines with integers of 32 bits or more.

Input Parameters

iseed	INTEGER. Array, size 4. On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and `iseed(4)` must be odd.

Output Parameters

iseed	INTEGER. On exit, the seed is updated.
res	`REAL` for slaran, `DOUBLE PRECISION` for dlaran, Random number.

?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call slarf( side, m, n, v, incv, tau, c, ldc, work )
call dlarf( side, m, n, v, incv, tau, c, ldc, work )
call clarf( side, m, n, v, incv, tau, c, ldc, work )
call zlarf( side, m, n, v, incv, tau, c, ldc, work )
```

Include Files

- `mkl.fi`

Description

The routine applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right. *H* is represented in one of the following forms:

- `H = I - tau*v*v^T`

  where *tau* is a real scalar and v is a real vector.

  If `tau = 0`, then *H* is taken to be the unit matrix.
- `H = I - tau*v*v^H`

  where *tau* is a complex scalar and v is a complex vector.

  If `tau = 0`, then *H* is taken to be the unit matrix. For `clarf`/`zlarf`, to apply H^H (the conjugate transpose of *H*), supply conjg(*tau*) instead of *tau*.

Input Parameters

side	`CHARACTER*1`. If `side = 'L'`: form *H***C* If `side = 'R'`: form *C***H*.
m	`INTEGER`. The number of rows of the matrix *C*.
n	`INTEGER`. The number of columns of the matrix *C*.

v	`REAL` for `slarf`
	`DOUBLE PRECISION` for `dlarf`
	`COMPLEX` for `clarf`
	`DOUBLE COMPLEX` for `zlarf`
	Array, `DIMENSION`
	`(1 + (m-1)*abs(incv))` if `side = 'L'` or
	`(1 + (n-1)*abs(incv))` if `side = 'R'`. The vector *v* in the representation of *H*. *v* is not used if *tau* = 0.

incv	`INTEGER`. The increment between elements of *v*. `incv≠ 0`.

tau	`REAL` for `slarf`
	`DOUBLE PRECISION` for `dlarf`
	`COMPLEX` for `clarf`
	`DOUBLE COMPLEX` for `zlarf`
	The value *tau* in the representation of *H*.

c	`REAL` for `slarf`
	`DOUBLE PRECISION` for `dlarf`
	`COMPLEX` for `clarf`
	`DOUBLE COMPLEX` for `zlarf`
	Array, `DIMENSION` (*ldc*,*n*).
	On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*.

ldc≥ max(1,*m*).

work REAL for slarf

DOUBLE PRECISION for dlarf

COMPLEX for clarf

DOUBLE COMPLEX for zlarf

Workspace array, DIMENSION

(*n*) if *side* = 'L' or

(*m*) if *side* = 'R'.

Output Parameters

c On exit, *C* is overwritten by the matrix *H***C* if *side* = 'L', or *C***H* if *side* = 'R'.

?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
call slarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work, ldwork )
call dlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work, ldwork )
call clarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work, ldwork )
call zlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work, ldwork )
```

Include Files

- mkl.fi

Description

The real flavors of the routine ?larfb apply a real block reflector *H* or its transpose H^T to a real *m*-by-*n* matrix *C* from either left or right.

The complex flavors of the routine ?larfb apply a complex block reflector *H* or its conjugate transpose H^H to a complex *m*-by-*n* matrix *C* from either left or right.

Input Parameters

The data types are given for the Fortran interface.

side CHARACTER*1.

If *side* = 'L': apply *H* or H^T for real flavors and *H* or H^H for complex flavors from the left.

If *side* = 'R': apply *H* or H^T for real flavors and *H* or H^H for complex flavors from the right.

trans CHARACTER*1.

If *trans* = 'N': apply *H* (No transpose).

If *trans* = 'C': apply H^H (Conjugate transpose).

If *trans* = 'T': apply H^T (Transpose).

direct CHARACTER*1.

Indicates how *H* is formed from a product of elementary reflectors

If *direct* = 'F': *H* = *H*(1)*H(2)*. . . *H*(*k*) (forward)

If *direct* = 'B': *H* = *H*(*k*)* . . . *H*(2)*H(1) (backward)

storev CHARACTER*1.

Indicates how the vectors which define the elementary reflectors are stored:

If *storev* = 'C': Column-wise

If *storev* = 'R': Row-wise

m INTEGER. The number of rows of the matrix *C*.

n INTEGER. The number of columns of the matrix *C*.

k INTEGER. The order of the matrix *T* (equal to the number of elementary reflectors whose product defines the block reflector).

v REAL for slarfb

DOUBLE PRECISION for dlarfb

COMPLEX for clarfb

DOUBLE COMPLEX for zlarfb

Array, DIMENSION

(*ldv*, *k*) if *storev* = 'C'

(*ldv*, *m*) if *storev* = 'R' and *side* = 'L'

(*ldv*, *n*) if *storev* = 'R' and *side* = 'R'

The matrix *v*. See *Application Notes* below.

ldv INTEGER. The leading dimension of the array v.

If *storev* = 'C' and *side* = 'L', *ldv*≥ max(1,*m*);

if *storev* = 'C' and *side* = 'R', *ldv*≥ max(1,*n*);

if *storev* = 'R', *ldv*≥*k*.

t REAL for slarfb

DOUBLE PRECISION for dlarfb

COMPLEX for clarfb

DOUBLE COMPLEX for zlarfb

Array, size (*ldt*,*k*).

Contains the triangular *k*-by-*k* matrix *T* in the representation of the block reflector.

ldt INTEGER. The leading dimension of the array *t*.

ldt≥*k*.

c REAL for slarfb

DOUBLE PRECISION for dlarfb

COMPLEX for clarfb

DOUBLE COMPLEX for zlarfb

Array, size (*ldc*,*n*).

On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*.

ldc≥ max(1,*m*).

work REAL for slarfb

DOUBLE PRECISION for dlarfb

COMPLEX for clarfb

DOUBLE COMPLEX for zlarfb

Workspace array, DIMENSION (*ldwork*, *k*).

ldwork INTEGER. The leading dimension of the array *work*.

If *side* = 'L', *ldwork*≥ max(1, *n*);

if *side* = 'R', *ldwork*≥ max(1, *m*).

Output Parameters

c On exit, *c* is overwritten by the product of the following:

- *H***C,* or H^T**C,* or *C***H,* or *C**H^T for real flavors
- *H***C,* or H^H**C,* or *C***H,* or *C**H^H for complex flavors

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = –*i*, the *i*-th parameter had an illegal value.

If *info* = –1011, memory allocation error occurred.

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the *H*(*i*) is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

$direct = 'F'$ and $storev = 'C'$:

$$\begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

$direct = 'F'$ and $storev = 'R'$:

$$\begin{bmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{bmatrix}$$

$direct = 'B'$ and $storev = 'C'$:

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

$direct = 'B'$ and $storev = 'R'$:

$$\begin{bmatrix} v_1 & v_1 & 1 & & \\ v_2 & v_2 & v_2 & 1 & \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call slarfg( n, alpha, x, incx, tau )
call dlarfg( n, alpha, x, incx, tau )
call clarfg( n, alpha, x, incx, tau )
call zlarfg( n, alpha, x, incx, tau )
```

Include Files

- mkl.fi

Description

The routine ?larfg generates a real/complex elementary reflector *H* of order *n*, such that

$$H * \begin{bmatrix} alpha \\ x \end{bmatrix} = \begin{bmatrix} beta \\ 0 \end{bmatrix}, \quad H^T * H = I,$$

for real flavors and

$$H^H * \begin{bmatrix} alpha \\ x \end{bmatrix} = \begin{bmatrix} beta \\ 0 \end{bmatrix}, \quad H^H * H = I,$$

for complex flavors,

where *alpha* and *beta* are scalars (with *beta* real for all flavors), and *x* is an (*n*-1)-element real/complex vector. *H* is represented in the form

$$H = I - tau * \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^T \end{bmatrix}$$

for real flavors and

$$H = I - tau * \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^H \end{bmatrix}$$

for complex flavors,

where *tau* is a real/complex scalar and v is a real/complex (*n*-1)-element vector, respectively. Note that for `clarfg`/`zlarfg`, *H* is not Hermitian.

If the elements of *x* are all zero (and, for complex flavors, *alpha* is real), then `tau = 0` and *H* is taken to be the unit matrix.

Otherwise, `1 ≤ tau ≤ 2` (for real flavors), or

`1 ≤ Re(tau) ≤ 2` and `abs(tau-1) ≤ 1` (for complex flavors).

Input Parameters

The data types are given for the Fortran interface.

n
: `INTEGER`. The order of the elementary reflector.

alpha
: `REAL` for `slarfg`
`DOUBLE PRECISION` for `dlarfg`
`COMPLEX` for `clarfg`
`DOUBLE COMPLEX` for `zlarfg` On entry, the value alpha.

x
: `REAL` for `slarfg`
`DOUBLE PRECISION` for `dlarfg`
`COMPLEX` for `clarfg`
`DOUBLE COMPLEX` for `zlarfg`
Array, size (1+(*n*-2)*abs(*incx*)).
On entry, the vector *x*.

incx
: `INTEGER`.
The increment between elements of *x*. `incx > 0`.

Output Parameters

alpha
: On exit, it is overwritten with the value *beta*.

x
: On exit, it is overwritten with the vector *v*.

tau
: `REAL` for `slarfg`
`DOUBLE PRECISION` for `dlarfg`
`COMPLEX` for `clarfg`
`DOUBLE COMPLEX` for `zlarfg` The value *tau*.

?larfgp

Generates an elementary reflector (Householder matrix) with non-negative beta .

Syntax

```
call slarfgp( n, alpha, x, incx, tau )
call dlarfgp( n, alpha, x, incx, tau )
call clarfgp( n, alpha, x, incx, tau )
call zlarfgp( n, alpha, x, incx, tau )
```

Include Files

- mkl.fi

Description

The routine ?larfgp generates a real/complex elementary reflector *H* of order *n*, such that

$$H * \begin{bmatrix} alpha \\ x \end{bmatrix} = \begin{bmatrix} beta \\ 0 \end{bmatrix}, \quad H^T * H = I,$$

for real flavors and

$$H^H * \begin{bmatrix} alpha \\ x \end{bmatrix} = \begin{bmatrix} beta \\ 0 \end{bmatrix}, \quad H^H * H = I,$$

for complex flavors,

where *alpha* and *beta* are scalars (with *beta* real and non-negative for all flavors), and *x* is an (*n*-1)-element real/complex vector. *H* is represented in the form

$$H = I - tau * \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^T \end{bmatrix}$$

for real flavors and

$$H = I - tau * \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^H \end{bmatrix}$$

for complex flavors,

where *tau* is a real/complex scalar and v is a real/complex (*n*-1)-element vector. Note that for c/zlarfgp, *H* is not Hermitian.

If the elements of *x* are all zero (and, for complex flavors, *alpha* is real), then $tau = 0$ and *H* is taken to be the unit matrix.

Otherwise, $1 \le tau \le 2$ (for real flavors), or

$1 \le \mathrm{Re}(tau) \le 2$ and $\mathrm{abs}(tau-1) \le 1$ (for complex flavors).

Input Parameters

n	INTEGER. The order of the elementary reflector.
alpha	REAL for slarfgp

DOUBLE PRECISION for dlarfgp

COMPLEX for clarfgp

DOUBLE COMPLEX for zlarfgp

On entry, the value alpha.

x

REAL for s

DOUBLE PRECISION for dlarfgp

COMPLEX for clarfgp

DOUBLE COMPLEX for zlarfgp

Array, DIMENSION (1+(*n*-2)*abs(*incx*)).

On entry, the vector *x*.

incx

INTEGER.

The increment between elements of *x*. *incx* > 0.

Output Parameters

alpha

On exit, it is overwritten with the value *beta*.

x

On exit, it is overwritten with the vector *v*.

tau

REAL for slarfgp

DOUBLE PRECISION for dlarfgp

COMPLEX for clarfgp

DOUBLE COMPLEX for zlarfgp

The value *tau*.

?larft

*Forms the triangular factor T of a block reflector H = I - V*T*V**H.*

Syntax

```
call slarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Include Files

- mkl.fi

Description

The routine ?larft forms the triangular factor *T* of a real/complex block reflector *H* of order *n*, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', *H* = *H*(1)*H(2)* . . .*H(k)* and *T* is upper triangular;

If *direct* = 'B', *H* = *H(k)*. . .*H(2)*H(1)* and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector H(*i*) is stored in the *i*-th column of the array *v*, and $H = I - V*T*V^T$ (for real flavors) or $H = I - V*T*V^H$ (for complex flavors) .

If *storev* = 'R', the vector which defines the elementary reflector H(*i*) is stored in the *i*-th row of the array *v*, and $H = I - V^T*T*V$ (for real flavors) or $H = I - V^H*T*V$ (for complex flavors).

Input Parameters

The data types are given for the Fortran interface.

direct

CHARACTER*1.

Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

= 'F': $H = H(1)*H(2)*. \ . \ . \ *H(k)$ (forward)

= 'B': $H = H(k)*. \ . \ .*H(2)*H(1)$ (backward)

storev

CHARACTER*1.

Specifies how the vectors which define the elementary reflectors are stored (see also *Application Notes* below):

= 'C': column-wise

= 'R': row-wise.

n

INTEGER. The order of the block reflector *H*. $n \geq 0$.

k

INTEGER. The order of the triangular factor *T* (equal to the number of elementary reflectors). $k \geq 1$.

v

REAL for slarft

DOUBLE PRECISION for dlarft

COMPLEX for clarft

DOUBLE COMPLEX for zlarft

Array, DIMENSION

(*ldv*, *k*) if *storev* = 'C' or

(*ldv*, *n*) if *storev* = 'R'.

The matrix *V*.

ldv

INTEGER. The leading dimension of the array *v*.

If *storev* = 'C', $ldv \geq \max(1,n)$

if *storev* = 'R', $ldv \geq k$.

tau

REAL for slarft

DOUBLE PRECISION for dlarft

COMPLEX for clarft

DOUBLE COMPLEX for zlarft

Array, size (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*).

ldt

INTEGER. The leading dimension of the output array *t*. $ldt \geq k$.

Output Parameters

t	`REAL` for `slarft`
	`DOUBLE PRECISION` for `dlarft`
	`COMPLEX` for `clarft`
	`DOUBLE COMPLEX` for `zlarft`
	Array, size *ldt* by *k*. The *k*-by-*k* triangular factor *T* of the block reflector. If *direct* = `'F'`, *T* is upper triangular; if *direct* = `'B'`, *T* is lower triangular. The rest of the array is not used.

v	The matrix *V*.

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the *H*(*i*) is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C':

$$\begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

direct = 'F' and *storev* = 'R':

$$\begin{bmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{bmatrix}$$

direct = 'B' and *storev* = 'C':

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

direct = 'B' and *storev* = 'R':

$$\begin{bmatrix} v_1 & v_1 & 1 & & \\ v_2 & v_2 & v_2 & 1 & \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfx

Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order less than or equal to 10.

Syntax

```
call slarfx( side, m, n, v, tau, c, ldc, work )
call dlarfx( side, m, n, v, tau, c, ldc, work )
call clarfx( side, m, n, v, tau, c, ldc, work )
call zlarfx( side, m, n, v, tau, c, ldc, work )
```

Include Files

- mkl.fi

Description

The routine ?larfx applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right.

H is represented in the following forms:

- *H = I - tau*v*v*^{T}, where *tau* is a real scalar and v is a real vector.
- *H = I - tau*v*v*^{H}, where *tau* is a complex scalar and v is a complex vector.

If *tau = 0*, then *H* is taken to be the unit matrix.

Input Parameters

The data types are given for the Fortran interface.

side — CHARACTER*1.

If *side = 'L'*: form *H*C*

If *side = 'R'*: form *C*H*.

m — INTEGER. The number of rows of the matrix *C*.

n — INTEGER. The number of columns of the matrix *C*.

v — REAL for slarfx

DOUBLE PRECISION for dlarfx

COMPLEX for clarfx

DOUBLE COMPLEX for zlarfx

Array, size

(*m*) if *side = 'L'* or

(*n*) if *side = 'R'*.

The vector *v* in the representation of *H*.

tau — REAL for slarfx

DOUBLE PRECISION for dlarfx

COMPLEX for clarfx

DOUBLE COMPLEX for zlarfx

The value *tau* in the representation of *H*.

c — REAL for slarfx

DOUBLE PRECISION for dlarfx

COMPLEX for clarfx

DOUBLE COMPLEX for zlarfx

Array, size *ldc* by *n*. On entry, the *m*-by-*n* matrix C.

ldc INTEGER. The leading dimension of the array *c*. *lda*≥ (1,*m*).

work REAL for slarfx

DOUBLE PRECISION for dlarfx

COMPLEX for clarfx

DOUBLE COMPLEX for zlarfx

Workspace array, size

(*n*) if *side* = 'L' or

(*m*) if *side* = 'R'.

work is not referenced if *H* has order < 11.

Output Parameters

c On exit, C is overwritten by the matrix *H*C* if *side* = 'L', or *C*H* if *side* = 'R'.

?larfy

Applies an elementary reflector, or Householder matrix, H, to an n by n symmetric or Hermitian matrix C, from both the left and the right.

```
call slarfy(uplo, n, v, incv, tau, C, ldc, work)
call dlarfy(uplo, n, v, incv, tau, C, ldc, work)
call clarfy(uplo, n, v, incv, tau, C, ldc, work)
call zlarfy(uplo, n, v, incv, tau, C, ldc, work)
```

Description

?larfy applies an elementary reflector, or Householder matrix, H, to an *n* by *n* symmetric or Hermitian matrix C, from both the left and the right. H is represented in the form H = I - *tau* * *v* * v^T (for real flavors) or H = I - *tau* * *v* * v^H (for complex flavors), where tau is a scalar and *v* is a vector. If *tau* is zero, H is taken to be the unit matrix.

Input Parameters

uplo CHARACTER*1

Specifies whether the upper or lower triangular part of the symmetric or Hermitian matrix C is stored.

- = 'U': Upper triangular part of C is stored.
- = 'L': Lower triangular part of C is stored.

n INTEGER

The number of rows and columns of the matrix C. *n* ≥ 0.

v REAL for slarfy

DOUBLE PRECISION for dlarfy

COMPLEX for clarfy

COMPLEX*16 for zlarfy

Array, dimension (1 + (*n*–1)*abs(*incv*)). The vector *v* as described above.

incv INTEGER

The increment between successive elements of v. *incv* must not be zero.

tau REAL for slarfy

DOUBLE PRECISION for dlarfy

COMPLEX for clarfy

COMPLEX*16 for zlarfy

The value *tau* as described above.

C REAL for slarfy

DOUBLE PRECISION for dlarfy

COMPLEX for clarfy

COMPLEX*16 for zlarfy

Array, dimension (*ldc*, *n*). On entry, the matrix C.

ldc INTEGER

The leading dimension of the array *C*. *ldc* ≥ max(1, *n*).

Output Parameters

C REAL for slarfy

DOUBLE PRECISION for dlarfy

COMPLEX for clarfy

COMPLEX*16 for zlarfy

On exit, *C* is overwritten by H * *C* * H^T (for real flavors) or H * *C* * H^H (for complex flavors).

work REAL for slarfy

DOUBLE PRECISION for dlarfy

COMPLEX for clarfy

COMPLEX*16 for zlarfy

Array, dimension (*n*).

?large

Pre- and post-multiplies a real general matrix with a random orthogonal matrix.

Syntax

```
call slarge( n, a, lda, iseed, work, info )
call dlarge( n, a, lda, iseed, work, info )
call clarge( n, a, lda, iseed, work, info )
```

```
call zlarge( n, a, lda, iseed, work, info )
```

Include Files

- mkl.fi

Description

The routine ?large pre- and post-multiplies a general *n*-by-*n* matrix *A* with a random orthogonal or unitary matrix: $A = U*D*U^T$.

Input Parameters

n — INTEGER. The order of the matrix *A*. $n \geq 0$

a — REAL for slarge,

DOUBLE PRECISION for dlarge,

COMPLEX for clarge,

DOUBLE COMPLEX for zlarge,

Array, size *lda* by *n*.

On entry, the original *n*-by-*n* matrix *A*.

lda — INTEGER. The leading dimension of the array *a*. $lda \geq n$.

iseed — INTEGER. Array, size 4.

On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and *iseed*(4) must be odd.

work — REAL for slarge,

DOUBLE PRECISION for dlarge,

COMPLEX for clarge,

DOUBLE COMPLEX for zlarge,

Workspace array, size 2*n.

Output Parameters

a — INTEGER.

On exit, *A* is overwritten by *U**A**U'* for some random orthogonal matrix *U*.

iseed — INTEGER.

On exit, the seed is updated.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0, the *i* -th parameter had an illegal value.

?largv

Generates a vector of plane rotations with real cosines and real/complex sines.

Syntax

```
call slargv( n, x, incx, y, incy, c, incc )
call dlargv( n, x, incx, y, incy, c, incc )
call clargv( n, x, incx, y, incy, c, incc )
call zlargv( n, x, incx, y, incy, c, incc )
```

Include Files

- mkl.fi

Description

The routine generates a vector of real/complex plane rotations with real cosines, determined by elements of the real/complex vectors *x* and *y*.

For slargv/dlargv:

$$\begin{bmatrix} c(i) & s(i) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} a_i \\ 0 \end{bmatrix}, \text{for } i = 1,2,\ldots,n$$

For clargv/zlargv:

$$\begin{bmatrix} c(i) & s(i) \\ -\text{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} r_i \\ 0 \end{bmatrix}, \text{for } i = 1,2,\ldots,n$$

where $c(i)^2 + \text{abs}(s(i))^2 = 1$ and the following conventions are used (these are the same as in clartg/zlartg but differ from the BLAS Level 1 routine crotg/zrotg):

If $y_i = 0$, then $c(i) = 1$ and $s(i) = 0$;

If $x_i = 0$, then $c(i) = 0$ and $s(i)$ is chosen so that r_i is real.

Input Parameters

n	INTEGER. The number of plane rotations to be generated.
x, *y*	REAL for slargv DOUBLE PRECISION for dlargv COMPLEX for clargv DOUBLE COMPLEX for zlargv Arrays, DIMENSION (1+(*n*-1)**incx*) and (1+(*n*-1)**incy*), respectively. On entry, the vectors *x* and *y*.

incx INTEGER. The increment between elements of *x*.

incx > 0.

incy INTEGER. The increment between elements of *y*.

incy > 0.

incc INTEGER. The increment between elements of the output array *c*. *incc* > 0.

Output Parameters

x On exit, $x(i)$ is overwritten by a_i (for real flavors), or by r_i (for complex flavors), for $i = 1, \ldots, n$.

y On exit, the sines $s(i)$ of the plane rotations.

c REAL for slargv/clargv

DOUBLE PRECISION for dlargv/zlargv

Array, DIMENSION (1+(*n*-1)**incc*). The cosines of the plane rotations.

?larnd

Returns a random real number from a uniform or normal distribution.

Syntax

```
res = slarnd( idist, iseed )
res = dlarnd( idist, iseed )
res = clarnd( idist, iseed )
res = zlarnd( idist, iseed )
```

Include Files

- mkl.fi

Description

The routine ?larnd returns a random number from a uniform or normal distribution.

Input Parameters

idist INTEGER. Specifies the distribution of the random numbers. For slarnd and dlanrd:

= 1: uniform (0,1)

= 2: uniform (-1,1)

= 3: normal (0,1).

For clarnd and zlanrd:

= 1: real and imaginary parts each uniform (0,1)

= 2: real and imaginary parts each uniform (-1,1)

= 3: real and imaginary parts each normal (0,1)

= 4: uniformly distributed on the disc abs(z) ≤ 1

= 5: uniformly distributed on the circle abs(z) = 1

iseed INTEGER. Array, size 4.

On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and *iseed*(4) must be odd.

Output Parameters

iseed INTEGER.

On exit, the seed is updated.

res REAL for slarnd,

DOUBLE PRECISION for dlarnd,

COMPLEX for clarnd,

DOUBLE COMPLEX for zlarnd,

Random number.

?larnv

Returns a vector of random numbers from a uniform or normal distribution.

Syntax

```
call slarnv( idist, iseed, n, x )
call dlarnv( idist, iseed, n, x )
call clarnv( idist, iseed, n, x )
call zlarnv( idist, iseed, n, x )
```

Include Files

- mkl.fi

Description

The routine ?larnv returns a vector of *n* random real/complex numbers from a uniform or normal distribution.

This routine calls the auxiliary routine ?laruv to generate random real numbers from a uniform (0,1) distribution, in batches of up to 128 using vectorisable code. The Box-Muller method is used to transform numbers from a uniform to a normal distribution.

Input Parameters

The data types are given for the Fortran interface.

idist INTEGER. Specifies the distribution of the random numbers: for slarnv and dlarnv:

= 1: uniform (0,1)

= 2: uniform (-1,1)

= 3: normal (0,1).

for `clarnv` and `zlarnv`:

= 1: real and imaginary parts each uniform (0,1)

= 2: real and imaginary parts each uniform (-1,1)

= 3: real and imaginary parts each normal (0,1)

= 4: uniformly distributed on the disc abs(z) < 1

= 5: uniformly distributed on the circle abs(z) = 1

iseed — `INTEGER`. Array, size (4).

On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and *iseed*(4) must be odd.

n — `INTEGER`. The number of random numbers to be generated.

Output Parameters

x — `REAL` for `slarnv`

`DOUBLE PRECISION` for `dlarnv`

`COMPLEX` for `clarnv`

`DOUBLE COMPLEX` for `zlarnv`

Array, size (*n*). The generated random numbers.

iseed — On exit, the seed is updated.

?laror

Pre- or post-multiplies an m-by-n matrix by a random orthogonal/unitary matrix.

Syntax

```
call slaror( side, init, m, n, a, lda, iseed, x, info )
call dlaror( side, init, m, n, a, lda, iseed, x, info )
call claror( side, init, m, n, a, lda, iseed, x, info )
call zlaror( side, init, m, n, a, lda, iseed, x, info )
```

Include Files

- `mkl.fi`

Description

The routine `?laror` pre- or post-multiplies an *m*-by-*n* matrix *A* by a random orthogonal or unitary matrix *U*, overwriting *A*. *A* may optionally be initialized to the identity matrix before multiplying by *U*. *U* is generated using the method of G.W. Stewart (SIAM J. Numer. Anal. 17, 1980, 403-409).

Input Parameters

side

CHARACTER*1. Specifies whether *A* is multiplied by *U* on the left or right.

for slaror and dlaror:

If *side* = 'L', multiply *A* on the left (premultiply) by *U*.

If *side* = 'R', multiply *A* on the right (postmultiply) by U^T.

If *side* = 'C' or 'T', multiply *A* on the left by *U* and the right by U^T.

for claror and zlaror:

If *side* = 'L', multiply *A* on the left (premultiply) by *U*.

If *side* = 'R', multiply *A* on the right (postmultiply) by *UC*>.

If *side* = 'C', multiply *A* on the left by *U* and the right by *UC*>

If*side* = 'T', multiply *A* on the left by *U* and the right by U^T.

init

CHARACTER*1. Specifies whether or not *a* should be initialized to the identity matrix.

If *init* = 'I', initialize *a* to (a section of) the identity matrix before applying *U*.

If *init* = 'N', no initialization. Apply *U* to the input matrix *A*.

init = 'I' generates square or rectangular orthogonal matrices:

For *m* = *n* and *side* = 'L' or 'R', the rows and the columns are orthogonal to each other.

For rectangular matrices where *m* < *n*:

- If *side* = 'R', ?laror produces a dense matrix in which rows are orthogonal and columns are not.
- If *side*= 'L', ?laror produces a matrix in which rows are orthogonal, first *m* columns are orthogonal, and remaining columns are zero.

For rectangular matrices where *m* > *n*:

- If *side* = 'L', ?laror produces a dense matrix in which columns are orthogonal and rows are not.
- If *side* = 'R', ?laror produces a matrix in which columns are orthogonal, first *m* rows are orthogonal, and remaining rows are zero.

m

INTEGER. The number of rows of *A*.

n

INTEGER. The number of columns of *A*.

a

REAL for slaror,

DOUBLE PRECISION for dlaror,

COMPLEX for claror,

DOUBLE COMPLEX for zlaror,

Array, size *lda* by *n*.

lda

INTEGER. The leading dimension of the array *a*.

lda≥ max(1, *m*).

iseed INTEGER. Array, size (4).

On entry, specifies the seed of the random number generator. The array elements must be between 0 and 4095; if not they are reduced mod 4096. Also, *iseed*(4) must be odd.

x REAL for slaror,

DOUBLE PRECISION for dlaror,

COMPLEX for claror,

DOUBLE COMPLEX for zlaror,

Workspace array, size (3*max(*m*, *n*)) .

Value of *side*	Length of workspace
'L'	2*m + n
'R'	2*n + m
'C' or 'T'	3*n

Output Parameters

a On exit, overwritten

by *UA* (if *side* = 'L'),

by *AU* (if *side* = 'R'),

by UAU^T (if *side* = 'C' or 'T').

iseed The values of *iseed* are changed on exit, and can be used in the next call to continue the same random number sequence.

info INTEGER. Array, size (4).

For slaror and dlaror:

If *info* = 0, the execution is successful.

If *info* < 0, the *i* -th parameter had an illegal value.

If *info* = 1, the random numbers generated by ?laror are bad.

For claror and zlaror:

If *info* = 0, the execution is successful.

If *info* = -1, *side* is not 'L', 'R', 'C', or 'T'.

If *info* = -3, if *m* is negative.

If *info* = -4, if *m* is negative or if side is 'C' or 'T' and *n* is not equal to *m* .

If *info* = -6, if *lda* is less than *m* .

?larot

Applies a Givens rotation to two adjacent rows or columns.

Syntax

```
call slarot( lrows, lleft, lright, nl, c, s, a, lda, xleft, xright )
call dlarot( lrows, lleft, lright, nl, c, s, a, lda, xleft, xright )
call clarot( lrows, lleft, lright, nl, c, s, a, lda, xleft, xright )
call zlarot( lrows, lleft, lright, nl, c, s, a, lda, xleft, xright )
```

Include Files

- mkl.fi

Description

The routine ?larot applies a Givens rotation to two adjacent rows or columns, where one element of the first or last column or row is stored in some format other than GE so that elements of the matrix may be used or modified for which no array element is provided.

One example is a symmetric matrix in SB format (bandwidth = 4), for which *uplo* = 'L'. Two adjacent rows will have the format:

```
row j :      C > C > C > C > C > . . . .
row j + 1 :  C > C > C > C > C > . . . .
```

'*' indicates elements for which storage is provided.

'.' indicates elements for which no storage is provided, but are not necessarily zero; their values are determined by symmetry.

' ' indicates elements which are required to be zero, and have no storage provided.

Those columns which have two '*' entries can be handled by srot (for slarot and clarot), or by drot(for dlarot and zlarot).

Those columns which have no '*' entries can be ignored, since as long as the Givens rotations are carefully applied to preserve symmetry, their values are determined.

Those columns which have one '*' have to be handled separately, by using separate variables *p* and *q* :

```
row j :      C > C > C > C > C > p. . . .
row j + 1 :  q  C > C > C > C > C > . . . .
```

If element *p* is set correctly, ?larot rotates the column and sets *p* to its new value. The next call to ?larot rotates columns *j* and *j* +1, and restore symmetry. The element *q* is zero at the beginning, and non-zero after the rotation. Later, rotations would presumably be chosen to zero *q* out.

Typical Calling Sequences: rotating the *i* -th and (*i* +1)–st rows.

Example

Typical Calling Sequences

These examples rotate the *i* -th and (*i* +1)–st rows.

General dense matrix:

```
call dlarot (.TRUE.,.FALSE.,.FALSE., n, c, s,
a(i,1),lda, dummy, dummy)
```

General banded matrix in GB format:

```
            j = max(1, i-kl )
            nl = min( n, i+ku+1 ) + 1-j
            call dlarot( .TRUE., i-kl.GE.1, i+ku.LT.n, nl, c,s,
                    a(ku+i+1-j,j),lda-1, xleft, xright )
```

NOTE
i + 1 - *j* is just min(*i*, *kl* + 1).

Symmetric banded matrix in SY format, bandwidth K, lower triangle only:

```
            j = max(1, i-k )
            nl = min( k+1, i ) + 1
            call dlarot( .TRUE., i-k.GE.1, .TRUE., nl, c,s,
                    a(i,j), lda, xleft, xright )
```

Same, but upper triangle only:

```
            nl = min( k+1, n-i ) + 1
            call dlarot( .TRUE., .TRUE., i+k.LT.n, nl, c,s,
                    a(i,i), lda, xleft, xright )
```

Symmetric banded matrix in SB format, bandwidth K, lower triangle only: [same as for SY, except:]

```
                . . . .
                    a(i+1-j,j), lda, xleft, xright )
```

NOTE
i+1-*j* is just min(*i*,*k*+1)

Same, but upper triangle only:

```
              . . . .
                    a(k+1,i), lda-1, xleft, xright )
```

Rotating columns is just the transpose of rotating rows, except for GB and SB: (rotating columns i and i+1)
GB:

NOTE
ku+*j*+1-*i* is just max(1,*ku*+2-*i*)

```
            j = max(1, i-ku )
            nl = min( n, i+kl+1 ) + 1-j
            call dlarot( .TRUE., i-ku.LE.1, i+kl.LT.n, nl, c,s,
                    a(ku+j+1-i,i),lda-1, xtop, xbottm )
```

SB: (upper triangle)

```
                . . . . . .
                    a(k+j+1-i,i),lda-1, xtop, xbottm )
```

SB: (lower triangle) . . . . . A(1,i),LDA-1, XTOP, XBOTTM)

```
                . . . . . .
                    a(1,i),lda-1, xtop, xbottm )
```

Input Parameters

lrows

LOGICAL.

If *lrows* = .TRUE., ?larot rotates two rows.

If *lrows* = .FALSE., ?larot rotates two columns.

lleft

LOGICAL.

If *lleft* = .TRUE., *xleft* is used instead of the corresponding element of *a* for the first element in the second row (if *lrows* = .FALSE.) or column (if *lrows*=.TRUE.).

If *lleft* = .FALSE., the corresponding element of *a* is used.

lright

LOGICAL.

If *lleft* = .TRUE., *xright* is used instead of the corresponding element of *a* for the first element in the second row (if *lrows* = .FALSE.) or column (if *lrows*=.TRUE.).

If *lright* = .FALSE., the corresponding element of *a* is used.

nl

INTEGER. The length of the rows (if *lrows*=.TRUE.) or columns (if *lrows*=.TRUE.) to be rotated.

If *xleft* or *xright* are used, the columns or rows they are in should be included in *nl*, e.g., if *lleft* = *lright* = .TRUE., then *nl* must be at least 2.

The number of rows or columns to be rotated exclusive of those involving *xleft* and/or *xright* may not be negative, i.e., *nl* minus how many of *lleft* and *lright* are .TRUE. must be at least zero; if not, xerbla is called.

c, s

REAL for slarot,

DOUBLE PRECISION for dlarot,

COMPLEX for clarot,

DOUBLE COMPLEX for zlarot,

Specify the Givens rotation to be applied.

If *lrows* = .TRUE., then the matrix

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

is applied from the left.

If *lrows* = .FALSE., then the transpose thereof is applied from the right.

a

REAL for slarot,

DOUBLE PRECISION for dlarot,

COMPLEX for clarot,

DOUBLE COMPLEX for zlarot,

The array containing the rows or columns to be rotated. The first element of *a* should be the upper left element to be rotated.

lda

INTEGER. The "effective" leading dimension of *a*.

If *a* contains a matrix stored in GE or SY format, then this is just the leading dimension of *A*.

If *a* contains a matrix stored in band (GB or SB) format, then this should be one less than the leading dimension used in the calling routine. Thus, if *a* is dimensioned *a*(*lda*,*) in ?larot, then *a*(1,*j*) would be the *j* -th element in the first of the two rows to be rotated, and *a*(2,*j*) would be the *j* -th in the second, regardless of how the array may be stored in the calling routine. *a* cannot be dimensioned, because for band format the row number may exceed *lda*, which is not legal FORTRAN.

If *lrows* = .TRUE., then *lda* must be at least 1, otherwise it must be at least *nl* minus the number of .TRUE. values in *xleft* and *xright*.

xleft

REAL for slarot,

DOUBLE PRECISION for dlarot,

COMPLEX for clarot,

DOUBLE COMPLEX for zlarot,

If *lrows* = .TRUE., *xleft* is used and modified instead of *a*(2,1) (if *lrows* = .TRUE.) or *a*(1,2) (if *lrows* = .FALSE.).

xright

REAL for slarot,

DOUBLE PRECISION for dlarot,

COMPLEX for clarot,

DOUBLE COMPLEX for zlarot,

If *lright* = .TRUE., *xright* is used and modified instead of *a*(1,*nl*) (if *lrows* = .TRUE.) or *a*(*nl*,1) (if *lrows* = .FALSE.).

Output Parameters

a

On exit, modified array *A*.

?larra

Computes the splitting points with the specified threshold.

Syntax

```
call slarra( n, d, e, e2, spltol, tnrm, nsplit, isplit, info )
call dlarra( n, d, e, e2, spltol, tnrm, nsplit, isplit, info )
```

Include Files

- mkl.fi

Description

The routine computes the splitting points with the specified threshold and sets any "small" off-diagonal elements to zero.

Input Parameters

n
INTEGER. The order of the matrix (*n*> 1).

d
REAL for slarra
DOUBLE PRECISION for dlarra
Array, DIMENSION (*n*).
Contains *n* diagonal elements of the tridiagonal matrix *T*.

e
REAL for slarra
DOUBLE PRECISION for dlarra
Array, DIMENSION (*n*).
First (*n*–1) entries contain the subdiagonal elements of the tridiagonal matrix *T*; *e*(*n*) need not be set.

e2
REAL for slarra
DOUBLE PRECISION for dlarra
Array, DIMENSION (*n*).
First (*n*–1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix *T*; *e2*(*n*) need not be set.

spltol
REAL for slarra
DOUBLE PRECISION for dlarra
The threshold for splitting. Two criteria can be used: *spltol*<0 : criterion based on absolute off-diagonal value; *spltol*>0 : criterion that preserves relative accuracy.

tnrm
REAL for slarra
DOUBLE PRECISION for dlarra
The norm of the matrix.

Output Parameters

e
On exit, the entries *e*(*isplit*(i)), 1 ≤ i ≤ *nsplit*, are set to zero, the other entries of *e* are untouched.

e2
On exit, the entries *e2*(*isplit*(i)), 1 ≤ i ≤ *nsplit*, are set to zero.

nsplit
INTEGER.
The number of blocks the matrix *T* splits into. 1 ≤ *nsplit* ≤ *n*

isplit
INTEGER.
Array, DIMENSION (*n*).

The splitting points, at which *T* breaks up into blocks. The first block consists of rows/columns `1` to *isplit*`(1)`, the second of rows/columns *isplit*`(1)+1` through *isplit*`(2)`, and so on, and the *nsplit*-th consists of rows/columns *isplit*`(`*nsplit*`-1)+1` through *isplit*`(`*nsplit*`)=`*n*.

info — INTEGER.

= 0: successful exit.

?larrb

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

```
call slarrb( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work, iwork, pivmin, spdiam, twist, info )
call dlarrb( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work, iwork, pivmin, spdiam, twist, info )
```

Include Files

- mkl.fi

Description

Given the relatively robust representation (RRR) $L*D*L^T$, the routine does "limited" bisection to refine the eigenvalues of $L*D*L^T$, *w*(*ifirst-offset*) through *w*(*ilast-offset*), to more accuracy. Initial guesses for these eigenvalues are input in *w*. The corresponding estimate of the error in these guesses and their gaps are input in *werr* and *wgap*, respectively. During bisection, intervals [*left*, *right*] are maintained by storing their mid-points and semi-widths in the arrays *w* and *werr* respectively.

Input Parameters

n — INTEGER. The order of the matrix.

d — REAL for slarrb

DOUBLE PRECISION for dlarrb

Array, DIMENSION (*n*). The *n* diagonal elements of the diagonal matrix *D*.

lld — REAL for slarrb

DOUBLE PRECISION for dlarrb

Array, DIMENSION (*n*-1).

The *n*-1 elements $L_i*L_i*D_i$.

ifirst — INTEGER. The index of the first eigenvalue to be computed.

ilast — INTEGER. The index of the last eigenvalue to be computed.

rtol1, *rtol2* — REAL for slarrb

DOUBLE PRECISION for dlarrb

Tolerance for the convergence of the bisection intervals. An interval [*left*, *right*] has converged if `RIGHT-LEFT.LT.MAX( rtol1*gap, rtol2*max(|left|,|right|) )`, where `gap` is the (estimated) distance to the nearest eigenvalue.

offset

`INTEGER`. Offset for the arrays *w*, *wgap* and *werr*, that is, the *ifirst-offset* through *ilast-offset* elements of these arrays are to be used.

w

`REAL` for `slarrb`

`DOUBLE PRECISION` for `dlarrb`

Array, `DIMENSION` (*n*). On input, *w*(*ifirst-offset*) through *w*(*ilast-offset*) are estimates of the eigenvalues of $L*D*L^T$ indexed *ifirst* through *ilast*.

wgap

`REAL` for `slarrb`

`DOUBLE PRECISION` for `dlarrb`

Array, `DIMENSION` (*n*-1). The estimated gaps between consecutive eigenvalues of $L*D*L^T$, that is, *wgap*(*i-offset*) is the gap between eigenvalues *i* and *i*+1. Note that if `IFIRST.EQ.ILAST` then *wgap*(*ifirst-offset*) must be set to 0.

werr

`REAL` for `slarrb`

`DOUBLE PRECISION` for `dlarrb`

Array, `DIMENSION` (*n*). On input, *werr*(*ifirst-offset*) through *werr*(*ilast-offset*) are the errors in the estimates of the corresponding elements in *w*.

work

`REAL` for `slarrb`

`DOUBLE PRECISION` for `dlarrb`

Workspace array, `DIMENSION` (2**n*).

pivmin

`REAL` for `slarrb`

`DOUBLE PRECISION` for `dlarrb`

The minimum pivot in the Sturm sequence.

spdiam

`REAL` for `slarrb`

`DOUBLE PRECISION` for `dlarrb`

The spectral diameter of the matrix.

twist

`INTEGER`. The twist index for the twisted factorization that is used for the negcount.

twist = *n*: Compute negcount from $L*D*L^T - lambda*i = L+* D+ *L+^T$
twist = *n*: Compute negcount from $L*D*L^T - lambda*i = U-*D-*U-^T$
twist = *n*: Compute negcount from $L*D*L^T - lambda*i = N_r*D_r*N_r$

iwork

`INTEGER`.

Workspace array, `DIMENSION` (2**n*).

Output Parameters

w	On output, the estimates of the eigenvalues are "refined".
wgap	On output, the gaps are refined.
werr	On output, "refined" errors in the estimates of *w*.
info	INTEGER. Error flag.

?larrc

Computes the number of eigenvalues of the symmetric tridiagonal matrix.

Syntax

```
call slarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
call dlarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
```

Include Files

- mkl.fi

Description

The routine finds the number of eigenvalues of the symmetric tridiagonal matrix *T* or of its factorization $L*D*L^T$ in the specified interval.

Input Parameters

jobt	CHARACTER*1. = 'T': computes Sturm count for matrix *T*. = 'L': computes Sturm count for matrix $L*D*L^T$.
n	INTEGER. The order of the matrix. ($n > 1$).
vl,*vu*	REAL for slarrc DOUBLE PRECISION for dlarrc The lower and upper bounds for the eigenvalues.
d	REAL for slarrc DOUBLE PRECISION for dlarrc Array, DIMENSION (*n*). If *jobt*= 'T': contains the *n* diagonal elements of the tridiagonal matrix *T*. If *jobt*= 'L': contains the *n* diagonal elements of the diagonal matrix *D*.
e	REAL for slarrc DOUBLE PRECISION for dlarrc

Array, DIMENSION (*n*).

If *jobt*= 'T': contains the (*n*-1) offdiagonal elements of the matrix *T*.

If *jobt*= 'L': contains the (*n*-1) offdiagonal elements of the matrix *L*.

pivmin — REAL for slarrc

DOUBLE PRECISION for dlarrc

The minimum pivot in the Sturm sequence for the matrix *T*.

Output Parameters

eigcnt — INTEGER.

The number of eigenvalues of the symmetric tridiagonal matrix *T* that are in the half-open interval (*vl*, *vu*].

lcnt,*rcnt* — INTEGER.

The left and right negcounts of the interval.

info — INTEGER.

Now it is not used and always is set to 0.

?larrd

Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Syntax

```
call slarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplit, isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )
```

```
call dlarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplit, isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )
```

Include Files

- mkl.fi

Description

The routine computes the eigenvalues of a symmetric tridiagonal matrix *T* to suitable accuracy. This is an auxiliary code to be called from ?stemr. The user may ask for all eigenvalues, all eigenvalues in the half-open interval (*vl*, *vu*], or the *il*-th through *iu*-th eigenvalues.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(overflow^{1/2}*underflow^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. (For more details see [Kahan66].

Input Parameters

range — CHARACTER.

= 'A': ("All") all eigenvalues will be found.

= 'V': ("Value") all eigenvalues in the half-open interval (*vl*, *vu*] will be found.

= 'I': ("Index") the *il*-th through *iu*-th eigenvalues will be found.

order CHARACTER.

= 'B': ("By block") the eigenvalues will be grouped by split-off block (see *iblock*, *isplit* below) and ordered from smallest to largest within the block.

= 'E': ("Entire matrix") the eigenvalues for the entire matrix will be ordered from smallest to largest.

n INTEGER. The order of the tridiagonal matrix *T* ($n \geq 1$).

vl,*vu* REAL for slarrd

DOUBLE PRECISION for dlarrd

If *range* = 'V': the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to *vl*, or greater than *vu*, will not be returned. $vl < vu$.

If *range* = 'A' or 'I': not referenced.

il,*iu* INTEGER.

If *range* = 'I': the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n=0$.

If *range* = 'A' or 'V': not referenced.

gers REAL for slarrd

DOUBLE PRECISION for dlarrd

Array, DIMENSION (2*n).

The *n* Gerschgorin intervals (the i-th Gerschgorin interval is (*gers*(2*i-1), *gers*(2*i)).

reltol REAL for slarrd

DOUBLE PRECISION for dlarrd

The minimum relative width of an interval. When an interval is narrower than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least *radix*machine epsilon*.

d REAL for slarrd

DOUBLE PRECISION for dlarrd

Array, DIMENSION (*n*).

Contains *n* diagonal elements of the tridiagonal matrix *T*.

e REAL for slarrd

DOUBLE PRECISION for dlarrd

Array, DIMENSION (*n*-1).

Contains (*n*-1) off-diagonal elements of the tridiagonal matrix *T*.

e2

REAL for slarrd

DOUBLE PRECISION for dlarrd

Array, DIMENSION (*n*-1).

Contains (*n*-1) squared off-diagonal elements of the tridiagonal matrix *T*.

pivmin

REAL for slarrd

DOUBLE PRECISION for dlarrd

The minimum pivot in the Sturm sequence for the matrix *T*.

nsplit

INTEGER.

The number of diagonal blocks the matrix *T* . $1 \le nsplit \le n$

isplit

INTEGER.

Arrays, DIMENSION (*n*).

The splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), and so on, and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.

(Only the first *nsplit* elements actually is used, but since the user cannot know a priori value of *nsplit*, *n* words must be reserved for *isplit*.)

work

REAL for slarrd

DOUBLE PRECISION for dlarrd

Workspace array, DIMENSION (4**n*).

iwork

INTEGER.

Workspace array, DIMENSION (4**n*).

Output Parameters

m

INTEGER.

The actual number of eigenvalues found. $0 \le m \le n$. (See also the description of *info*=2, 3.)

w

REAL for slarrd

DOUBLE PRECISION for dlarrd

Array, DIMENSION (*n*).

The first *m* elements of *w* contain the eigenvalue approximations. ?laprd computes an interval $I_j = (a_j, b_j]$ that includes eigenvalue j. The eigenvalue approximation is given as the interval midpoint $w(j) = (a_j + b_j)/2$. The corresponding error is bounded by $werr(j) = abs(a_j - b_j)/2$.

werr

REAL for slarrd

DOUBLE PRECISION for dlarrd

Array, DIMENSION (*n*).

The error bound on the corresponding eigenvalue approximation in *w*.

wl, *wu*

REAL for slarrd

DOUBLE PRECISION for dlarrd

The interval (*wl*, *wu*] contains all the wanted eigenvalues.

If *range* = 'V': then *wl*=*vl* and *wu*=*vu*.

If *range* = 'A': then *wl* and *wu* are the global Gerschgorin bounds on the spectrum.

If *range* = 'I': then *wl* and *wu* are computed by ?laebz from the index range specified.

iblock

INTEGER.

Array, DIMENSION (*n*).

At each row/column j where *e*(j) is zero or small, the matrix *T* is considered to split into a block diagonal matrix.

If *info* = 0, then *iblock*(i) specifies to which block (from 1 to the number of blocks) the eigenvalue *w*(i) belongs. (The routine may use the remaining *n*–*m* elements as workspace.)

indexw

INTEGER.

Array, DIMENSION (*n*).

The indices of the eigenvalues within each block (submatrix); for example, *indexw*(i)= j and *iblock*(i)=k imply that the i-th eigenvalue *w*(i) is the j-th eigenvalue in block k.

info

INTEGER.

= 0: successful exit.

< 0: if *info* = -i, the i-th argument has an illegal value

> 0: some or all of the eigenvalues fail to converge or are not computed:

=1 or 3: bisection fail to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.

=2 or 3:*range*='I' only: not all of the eigenvalues *il*:*iu* are found.

=4: *range*='I', and the Gershgorin interval initially used is too small. No eigenvalues are computed.

?larre

Given the tridiagonal matrix T, sets small off-diagonal elements to zero and for each unreduced block T_i, finds base representations and eigenvalues.

Syntax

```
call slarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit, m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

```
call dlarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit, m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

Include Files

- mkl.fi

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T, the routine sets any "small" off-diagonal elements to zero, and for each unreduced block T_i, it finds

- a suitable shift at one end of the block spectrum
- the base representation, $T_i - \sigma_i*I = L_i*D_i*L_i^T$, and
- eigenvalues of each $L_i*D_i*L_i^T$.

The representations and eigenvalues found are then used by ?stemr to compute the eigenvectors of a symmetric tridiagonal matrix. The accuracy varies depending on whether bisection is used to find a few eigenvalues or the dqds algorithm (subroutine ?lasq2) to compute all and discard any unwanted one. As an added benefit, ?larre also outputs the n Gerschgorin intervals for the matrices $L_i*D_i*L_i^T$.

Input Parameters

range

CHARACTER.

= 'A': ("All") all eigenvalues will be found.

= 'V': ("Value") all eigenvalues in the half-open interval (*vl*, *vu*] will be found.

= 'I': ("Index") the *il*-th through *iu*-th eigenvalues of the entire matrix will be found.

n

INTEGER. The order of the matrix. $n > 0$.

vl, *vu*

REAL for slarre

DOUBLE PRECISION for dlarre

If *range*='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to *vl*, or greater than *vu*, are not returned. $vl < vu$.

il, *iu*

INTEGER.

If *range*='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \le il \le iu \le n$.

d

REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (n).

The n diagonal elements of the diagonal matrices T.

e

REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (*n*). The first (*n*-1) entries contain the subdiagonal elements of the tridiagonal matrix *T*; *e*(*n*) need not be set.

e2

REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (*n*). The first (*n*-1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix *T*; *e2*(*n*) need not be set.

rtol1, *rtol2*

REAL for slarre

DOUBLE PRECISION for dlarre

Parameters for bisection. An interval [LEFT,RIGHT] has converged if RIGHT-LEFT.LT.MAX(*rtol1**gap, *rtol2**max(|LEFT|,|RIGHT|)).

spltol

REAL for slarre

DOUBLE PRECISION for dlarre

The threshold for splitting.

work

REAL for slarre

DOUBLE PRECISION for dlarre

Workspace array, DIMENSION (6**n*).

iwork

INTEGER.

Workspace array, DIMENSION (5**n*).

Output Parameters

vl, *vu*

On exit, if *range*='I' or ='A', contain the bounds on the desired part of the spectrum.

d

On exit, the *n* diagonal elements of the diagonal matrices D_i .

e

On exit, the subdiagonal elements of the unit bidiagonal matrices L_i . The entries *e*(*isplit*(*i*)), 1 ≤ *i* ≤ *nsplit*, contain the base points $sigma_i$ on output.

e2

On exit, the entries *e2*(*isplit*(*i*)), 1 ≤ *i* ≤ *nsplit*, have been set to zero.

nsplit

INTEGER. The number of blocks *T* splits into. 1 ≤ *nsplit* ≤ *n*.

isplit

INTEGER. Array, DIMENSION (*n*). The splitting points, at which *T* breaks up into blocks. The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.

m

INTEGER. The total number of eigenvalues (of all the $L_i*D_i*L_i^T$) found.

w

REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (*n*). The first *m* elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i*D_i*L_i^T$, are sorted in ascending order. The routine may use the remaining *n*-*m* elements as workspace.

werr

REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (*n*). The error bound on the corresponding eigenvalue in *w*.

wgap

REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (*n*). The separation from the right neighbor eigenvalue in *w*. The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree. Exception: at the right end of a block the left gap is stored.

iblock

INTEGER. Array, DIMENSION (*n*).

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in *w*; *iblock*(*i*)=1 if eigenvalue *w*(*i*) belongs to the first block from the top, =2 if *w*(*i*) belongs to the second block, etc.

indexw

INTEGER. Array, DIMENSION (*n*).

The indices of the eigenvalues within each block (submatrix); for example, *indexw*(*i*)= 10 and *iblock*(*i*)=2 imply that the *i*-th eigenvalue *w*(*i*) is the 10-th eigenvalue in the second block.

gers

REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (2*n*). The *n* Gerschgorin intervals (the *i*-th Gerschgorin interval is (*gers*(2*i*-1), *gers*(2*i*)).

pivmin

REAL for slarre

DOUBLE PRECISION for dlarre

The minimum pivot in the Sturm sequence for *T* .

info

INTEGER.

If *info* = 0: successful exit

If *info* > 0: A problem occured in ?larre. If *info* = 5, the Rayleigh Quotient Iteration failed to converge to full accuracy.

If *info* < 0: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter *info* for further information is required.

- If *info* = -1, there is a problem in ?larrd
- If *info* = -2, no base representation could be found in maxtry iterations. Increasing maxtry and recompilation might be a remedy.
- If *info* = -3, there is a problem in ?larrb when computing the refined root representation for ?lasq2.

- If *info* = -4, there is a problem in ?larrb when preforming bisection on the desired part of the spectrum.
- If *info* = -5, there is a problem in ?lasq2.
- If *info* = -6, there is a problem in ?lasq2.

See Also

?stemr

?lasq2

?larrb

?larrd

?larrf

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

```
call slarrf( n, d, l, ld, clstrt, clend, w, wgap, werr, spdiam, clgapl, clgapr, pivmin, sigma, dplus, lplus, work, info )
```

```
call dlarrf( n, d, l, ld, clstrt, clend, w, wgap, werr, spdiam, clgapl, clgapr, pivmin, sigma, dplus, lplus, work, info )
```

Include Files

- mkl.fi

Description

Given the initial representation $L*D*L^T$ and its cluster of close eigenvalues (in a relative measure), *w*(*clstrt*), *w*(*clstrt*+1), ... *w*(*clend*), the routine ?larrf finds a new relatively robust representation

$$L*D*L^T - \sigma_i*I = L(+)*D(+)*L(+)^T$$

such that at least one of the eigenvalues of $L(+)*D*(+)*L(+)^T$ is relatively isolated.

Input Parameters

n

INTEGER. The order of the matrix (subblock, if the matrix is splitted).

d

REAL for slarrf

DOUBLE PRECISION for dlarrf

Array, DIMENSION (*n*). The *n* diagonal elements of the diagonal matrix *D*.

l

REAL for slarrf

DOUBLE PRECISION for dlarrf

Array, DIMENSION (*n*-1).

The (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L*.

ld

REAL for slarrf

DOUBLE PRECISION for dlarrf

Array, DIMENSION (*n*-1).

The n-1 elements L_i*D_i.

clstrt INTEGER. The index of the first eigenvalue in the cluster.

clend INTEGER. The index of the last eigenvalue in the cluster.

w REAL for slarrf

DOUBLE PRECISION for dlarrf

Array, DIMENSION ≥ (*clend* -*clstrt*+1). The eigenvalue approximations of $L*D*L^T$ in ascending order. *w*(*clstrt*) through *w*(*clend*) form the cluster of relatively close eigenvalues.

wgap REAL for slarrf

DOUBLE PRECISION for dlarrf

Array, DIMENSION ≥ (*clend* -*clstrt*+1). The separation from the right neighbor eigenvalue in *w*.

werr REAL for slarrf

DOUBLE PRECISION for dlarrf

Array, DIMENSION ≥ (*clend* -*clstrt*+1). On input, *werr* contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in *w*.

spdiam REAL for slarrf

DOUBLE PRECISION for dlarrf

Estimate of the spectral diameter obtained from the Gerschgorin intervals.

clgapl, *clgapr* REAL for slarrf

DOUBLE PRECISION for dlarrf

Absolute gap on each end of the cluster. Set by the calling routine to protect against shifts too close to eigenvalues outside the cluster.

pivmin REAL for slarrf

DOUBLE PRECISION for dlarrf

The minimum pivot allowed in the Sturm sequence.

work REAL for slarrf

DOUBLE PRECISION for dlarrf

Workspace array, DIMENSION (2*n).

Output Parameters

wgap On output, the gaps are refined.

sigma REAL for slarrf

DOUBLE PRECISION for dlarrf

The shift used to form $L(+)*D*(+)*L(+)^T$.

dplus REAL for slarrf

DOUBLE PRECISION for dlarrf

Array, DIMENSION (*n*). The *n* diagonal elements of the diagonal matrix *D*(+).

lplus — REAL for slarrf

DOUBLE PRECISION for dlarrf

Array, DIMENSION (*n*). The first (*n*-1) elements of *lplus* contain the subdiagonal elements of the unit bidiagonal matrix *L*(+).

?larrj

Performs refinement of the initial estimates of the eigenvalues of the matrix T.

Syntax

```
call slarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork, pivmin, spdiam, info )
call dlarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork, pivmin, spdiam, info )
```

Include Files

- mkl.fi

Description

Given the initial eigenvalue approximations of *T*, this routine does bisection to refine the eigenvalues of *T*, *w*(*ifirst-offset*) through *w*(*ilast-offset*), to more accuracy. Initial guesses for these eigenvalues are input in *w*, the corresponding estimate of the error in these guesses in *werr*. During bisection, intervals [a,b] are maintained by storing their mid-points and semi-widths in the arrays *w* and *werr* respectively.

Input Parameters

n — INTEGER. The order of the matrix *T*.

d — REAL for slarrj

DOUBLE PRECISION for dlarrj

Array, DIMENSION (*n*).

Contains *n* diagonal elements of the matrix *T*.

e2 — REAL for slarrj

DOUBLE PRECISION for dlarrj

Array, DIMENSION (*n*-1).

Contains (*n*-1) squared sub-diagonal elements of the *T*.

ifirst — INTEGER.

The index of the first eigenvalue to be computed.

ilast — INTEGER.

The index of the last eigenvalue to be computed.

rtol

REAL for slarrj

DOUBLE PRECISION for dlarrj

Tolerance for the convergence of the bisection intervals. An interval [a,b] is considered to be converged if (b-a) ≤ *rtol**max(|a|,|b|).

offset

INTEGER.

Offset for the arrays *w* and *werr*, that is the *ifirst-offset* through *ilast-offset* elements of these arrays are to be used.

w

REAL for slarrj

DOUBLE PRECISION for dlarrj

Array, DIMENSION (*n*).

On input, *w*(*ifirst-offset*) through *w*(*ilast-offset*) are estimates of the eigenvalues of $L*D*L^T$ indexed *ifirst* through *ilast*.

werr

REAL for slarrj

DOUBLE PRECISION for dlarrj

Array, DIMENSION (*n*).

On input, *werr*(*ifirst-offset*) through *werr*(*ilast-offset*) are the errors in the estimates of the corresponding elements in *w*.

work

REAL for slarrj

DOUBLE PRECISION for dlarrj

Workspace array, DIMENSION (2**n*).

iwork

INTEGER.

Workspace array, DIMENSION (2**n*).

pivmin

REAL for slarrj

DOUBLE PRECISION for dlarrj

The minimum pivot in the Sturm sequence for the matrix *T*.

spdiam

REAL for slarrj

DOUBLE PRECISION for dlarrj

The spectral diameter of the matrix *T*.

Output Parameters

w

On exit, contains the refined estimates of the eigenvalues.

werr

On exit, contains the refined errors in the estimates of the corresponding elements in *w*.

info

INTEGER.

Now it is not used and always is set to 0.

?larrk

Computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy.

Syntax

```
call slarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
call dlarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
```

Include Files

- mkl.fi

Description

The routine computes one eigenvalue of a symmetric tridiagonal matrix *T* to suitable accuracy. This is an auxiliary code to be called from ?stemr.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(overflow^{1/2}*underflow^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. For more details see [Kahan66].

Input Parameters

n
INTEGER. The order of the matrix *T*. ($n \geq 1$).

iw
INTEGER.
The index of the eigenvalue to be returned.

gl, *gu*
REAL for slarrk
DOUBLE PRECISION for dlarrk
An upper and a lower bound on the eigenvalue.

d
REAL for slarrk
DOUBLE PRECISION for dlarrk
Array, DIMENSION (*n*).
Contains *n* diagonal elements of the matrix *T*.

e2
REAL for slarrk
DOUBLE PRECISION for dlarrk
Array, DIMENSION (*n*-1).
Contains (*n*-1) squared off-diagonal elements of the *T*.

pivmin
REAL for slarrk
DOUBLE PRECISION for dlarrk
The minimum pivot in the Sturm sequence for the matrix *T*.

reltol
REAL for slarrk
DOUBLE PRECISION for dlarrk

The minimum relative width of an interval. When an interval is narrower than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least *radix*machine epsilon*.

Output Parameters

w

REAL for slarrk

DOUBLE PRECISION for dlarrk

Contains the eigenvalue approximation.

werr

REAL for slarrk

DOUBLE PRECISION for dlarrk

Contains the error bound on the corresponding eigenvalue approximation in *w*.

info

INTEGER.

= 0: Eigenvalue converges

= -1: Eigenvalue does not converge

?larrr

Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Syntax

```
call slarrr( n, d, e, info )
call dlarrr( n, d, e, info )
```

Include Files

- mkl.fi

Description

The routine performs tests to decide whether the symmetric tridiagonal matrix *T* warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Input Parameters

n

INTEGER. The order of the matrix *T*. (*n*> 0).

d

REAL for slarrr

DOUBLE PRECISION for dlarrr

Array, DIMENSION (*n*).

Contains *n* diagonal elements of the matrix *T*.

e

REAL for slarrr

DOUBLE PRECISION for dlarrr

Array, DIMENSION (*n*).

The first (*n*-1) entries contain sub-diagonal elements of the tridiagonal matrix *T*; e(*n*) is set to 0.

Output Parameters

info INTEGER.

= 0: the matrix warrants computations preserving relative accuracy (default value).

= -1: the matrix warrants computations guaranteeing only absolute accuracy.

?larrv

*Computes the eigenvectors of the tridiagonal matrix T = L*D*L^T given L, D and the eigenvalues of L*D*L^T.*

Syntax

```
call slarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
call dlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
call clarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
call zlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
```

Include Files

- mkl.fi

Description

The routine ?larrv computes the eigenvectors of the tridiagonal matrix $T = L*D*L^T$ given *L*, *D* and approximations to the eigenvalues of $L*D*L^T$.

The input eigenvalues should have been computed by slarre for real flavors (slarrv/clarrv) and by dlarre for double precision flavors (dlarre/zlarre).

Input Parameters

n INTEGER. The order of the matrix. *n* ≥ 0.

vl, *vu* REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Lower and upper bounds respectively of the interval that contains the desired eigenvalues. *vl* < *vu*. Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range.

d

REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Array, DIMENSION (*n*). On entry, the *n* diagonal elements of the diagonal matrix *D*.

l

REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Array, DIMENSION (*n*).

On entry, the (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L* are contained in elements 1 to *n*-1 of *L* if the matrix is not splitted. At the end of each block the corresponding shift is stored as given by slarre for real flavors and by dlarre for double precision flavors.

pivmin

REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

The minimum pivot allowed in the Sturm sequence.

isplit

INTEGER. Array, DIMENSION (*n*).

The splitting points, at which *T* breaks up into blocks. The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc.

m

INTEGER. The total number of eigenvalues found.

$0 \le m \le n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu - il + 1$.

dol, *dou*

INTEGER.

If you want to compute only selected eigenvectors from all the eigenvalues supplied, specify an index range *dol*:*dou*. Or else apply the setting *dol*=1, *dou*=*m*. Note that *dol* and *dou* refer to the order in which the eigenvalues are stored in *w*.

If you want to compute only selected eigenpairs, then the columns *dol*-1 to *dou*+1 of the eigenvector space *Z* contain the computed eigenvectors. All other columns of *Z* are set to zero.

minrgp, *rtol1*, *rtol2*

REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Parameters for bisection. An interval [LEFT,RIGHT] has converged if RIGHT-LEFT.LT.MAX(*rtol1**gap, *rtol2**max(|LEFT|,|RIGHT|)).

w

REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Array, DIMENSION (*n*). The first *m* elements of *w* contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block (the output array *w* from ?larre is expected here). These eigenvalues are set with respect to the shift of the corresponding root representation for their block.

werr
REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Array, DIMENSION (*n*). The first *m* elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in *w*.

wgap
REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Array, DIMENSION (*n*). The separation from the right neighbor eigenvalue in *w*.

iblock
INTEGER. Array, DIMENSION (*n*).
The indices of the blocks (submatrices) associated with the corresponding eigenvalues in *w*; *iblock*(*i*)=1 if eigenvalue *w*(*i*) belongs to the first block from the top, =2 if *w*(*i*) belongs to the second block, etc.

indexw
INTEGER. Array, DIMENSION (*n*).
The indices of the eigenvalues within each block (submatrix); for example, *indexw*(*i*)= 10 and *iblock*(*i*)=2 imply that the *i*-th eigenvalue *w*(*i*) is the 10-th eigenvalue in the second block.

gers
REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Array, DIMENSION (2**n*). The *n* Gerschgorin intervals (the *i*-th Gerschgorin interval is (*gers*(2**i*-1), *gers*(2**i*)). The Gerschgorin intervals should be computed from the original unshifted matrix.

ldz
INTEGER. The leading dimension of the output array *Z*. *ldz*≥ 1, and if *jobz* = 'V', *ldz*≥ max(1,*n*).

work
REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Workspace array, DIMENSION (12**n*).

iwork
INTEGER.
Workspace array, DIMENSION (7**n*).

Output Parameters

d
On exit, *d* may be overwritten.

l
On exit, *l* is overwritten.

w
On exit, *w* holds the eigenvalues of the unshifted matrix.

werr
On exit, *werr* contains refined values of its input approximations.

wgap
On exit, *wgap* contains refined values of its input approximations. Very small gaps are changed.

z
REAL for slarrv
DOUBLE PRECISION for dlarrv

`COMPLEX` for `clarrv`

`DOUBLE COMPLEX` for `zlarrv`

Array, `DIMENSION` (*ldz*, max(1,*m*)).

If *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the input eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

NOTE
The user must ensure that at least max(1,*m*) columns are supplied in the array *z*.

isuppz

`INTEGER` .

Array, `DIMENSION(2*max(1,m))`. The support of the eigenvectors in *z*, that is, the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*).

info

`INTEGER`.

If *info* = 0: successful exit

If *info* > 0: A problem occured in `?larrv`. If *info* = 5, the Rayleigh Quotient Iteration failed to converge to full accuracy.

If *info* < 0: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter *info* for further information is required.

- If *info* = -1, there is a problem in `?larrb` when refining a child eigenvalue;
- If *info* = -2, there is a problem in `?larrf` when computing the relatively robust representation (RRR) of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter *minrgp* smaller and recompile. However, as the orthogonality of the computed vectors is proportional to 1/*minrgp*, you should be aware that you might be trading in precision when you decrease *minrgp*.
- If *info* = -3, there is a problem in `?larrb` when refining a single eigenvalue after the Rayleigh correction was rejected.

See Also

`?larrb`

`?larre`

`?larrf`

?lartg

Generates a plane rotation with real cosine and real/ complex sine.

Syntax

```
call slartg( f, g, cs, sn, r )
call dlartg( f, g, cs, sn, r )
```

```
call clartg( f, g, cs, sn, r )
call zlartg( f, g, cs, sn, r )
```

Include Files

- mkl.fi

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -conjg(sn) & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + |sn|^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine ?rotg, except for the following differences.

For slartg/dlartg:

f and *g* are unchanged on return;

If $g=0$, then *cs*=1 and *sn*=0;

If $f=0$ and $g \neq 0$, then $cs=0$ and $sn=1$ without doing any floating point operations (saves work in ?bdsqr when there are zeros on the diagonal);

If *f* exceeds *g* in magnitude, *cs* will be positive.

For clartg/zlartg:

f and *g* are unchanged on return;

If $g=0$, then $cs=1$ and $sn=0$;

If $f=0$, then $cs=0$ and *sn* is chosen so that *r* is real.

Input Parameters

f, *g*

REAL for slartg

DOUBLE PRECISION for dlartg

COMPLEX for clartg

DOUBLE COMPLEX for zlartg

The first and second component of vector to be rotated.

Output Parameters

cs

REAL for slartg/clartg

DOUBLE PRECISION for dlartg/zlartg

The cosine of the rotation.

sn	REAL for slartg DOUBLE PRECISION for dlartg COMPLEX for clartg DOUBLE COMPLEX for zlartg The sine of the rotation.
r	REAL for slartg DOUBLE PRECISION for dlartg COMPLEX for clartg DOUBLE COMPLEX for zlartg The nonzero component of the rotated vector.

?lartgp

Generates a plane rotation.

Syntax

```
call slartgp( f, g, cs, sn, r )
call dlartgp( f, g, cs, sn, r )
call lartgp( f,g,cs,sn,r )
```

Include Files

- mkl.fi

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + sn^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine ?rotg, except for the following differences:

- *f* and *g* are unchanged on return.
- If *g*=0, then *cs*=(+/-)1 and *sn*=0.
- If *f*=0 and *g*≠ 0, then *cs*=0 and *sn*=(+/-)1.

The sign is chosen so that *r*≥ 0.

Input Parameters

The data types are given for the Fortran interface.

f, *g*

REAL for slartgp

DOUBLE PRECISION for dlartgp

The first and second component of the vector to be rotated.

Output Parameters

cs

REAL for slartgp

DOUBLE PRECISION for dlartgp

The cosine of the rotation.

sn

REAL for slartgp

DOUBLE PRECISION for dlartgp

The sine of the rotation.

r

REAL for slartgp

DOUBLE PRECISION for dlartgp

The nonzero component of the rotated vector.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* =-1,*f* is NaN.

If *info* = -2, *g* is NaN.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ?lartgp interface are as follows:

f Holds the first component of the vector to be rotated.

g Holds the second component of the vector to be rotated.

cs Holds the cosine of the rotation.

sn Holds the sine of the rotation.

r Holds the nonzero component of the rotated vector.

See Also

?rotg

?lartg

?lartgs

?lartgs

Generates a plane rotation designed to introduce a bulge in implicit QR iteration for the bidiagonal SVD problem.

Syntax

```
call slartgs( x, y, sigma, cs, sn )
```

```
call dlartgs( x, y, sigma, cs, sn )
call lartgs( x,y,sigma,cs,sn )
```

Include Files

- mkl.fi

Description

The routine generates a plane rotation designed to introduce a bulge in Golub-Reinsch-style implicit QR iteration for the bidiagonal SVD problem. *x* and *y* are the top-row entries, and *sigma* is the shift. The computed *cs* and *sn* define a plane rotation that satisfies the following:

$$\begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \cdot \begin{bmatrix} x^2 - sigma \\ x * y \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

with *r* nonnegative.

If x^2 – *sigma* and *x* * *y* are 0, the rotation is by $\pi/2$

Input Parameters

The data types are given for the Fortran interface.

x, *y*
REAL for slartgs
DOUBLE PRECISION for dlartgs
The (1,1) and (1,2) entries of an upper bidiagonal matrix, respectively.

sigma
REAL for slartgs
DOUBLE PRECISION for dlartgs
Shift

Output Parameters

cs
REAL for slartgs
DOUBLE PRECISION for dlartgs
The cosine of the rotation.

sn
REAL for slartgs
DOUBLE PRECISION for dlartgs
The sine of the rotation.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine ?lartgs interface are as follows:

x	Holds the (1,1) entry of an upper diagonal matrix.
y	Holds the (1,2) entry of an upper diagonal matrix.
sigma	Holds the shift.
cs	Holds the cosine of the rotation.
sn	Holds the sine of the rotation.

See Also

?lartg

?lartgp

?lartv

Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.

Syntax

```
call slartv( n, x, incx, y, incy, c, s, incc )
call dlartv( n, x, incx, y, incy, c, s, incc )
call clartv( n, x, incx, y, incy, c, s, incc )
call zlartv( n, x, incx, y, incy, c, s, incc )
```

Include Files

- mkl.fi

Description

The routine applies a vector of real/complex plane rotations with real cosines to elements of the real/complex vectors *x* and *y*. For `i = 1,2,...,n`

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} := \begin{bmatrix} c(i) & s(i) \\ -\mathrm{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Input Parameters

n	INTEGER. The number of plane rotations to be applied.
x, *y*	REAL for slartv DOUBLE PRECISION for dlartv COMPLEX for clartv DOUBLE COMPLEX for zlartv

	Arrays, DIMENSION (1+(*n*-1)***incx*) and (1+(*n*-1)***incy*), respectively. The input vectors *x* and *y*.
incx	INTEGER. The increment between elements of *x*. *incx* > 0.
incy	INTEGER. The increment between elements of *y*. *incy* > 0.
c	REAL for slartv/clartv DOUBLE PRECISION for dlartv/zlartv Array, DIMENSION (1+(*n*-1)***incc*). The cosines of the plane rotations.
s	REAL for slartv DOUBLE PRECISION for dlartv COMPLEX for clartv DOUBLE COMPLEX for zlartv Array, DIMENSION (1+(*n*-1)***incc*). The sines of the plane rotations.
incc	INTEGER. The increment between elements of *c* and *s*. *incc* > 0.

Output Parameters

x, *y*	The rotated vectors *x* and *y*.

?laruv

Returns a vector of n random real numbers from a uniform distribution.

Syntax

```
call slaruv( iseed, n, x )
call dlaruv( iseed, n, x )
```

Include Files

- mkl.fi

Description

The routine ?laruv returns a vector of *n* random real numbers from a uniform (0,1) distribution (n ≤ 128).

This is an auxiliary routine called by ?larnv.

Input Parameters

iseed	INTEGER. Array, DIMENSION (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and *iseed*(4) must be odd.
n	INTEGER. The number of random numbers to be generated. *n* ≤ 128.

Output Parameters

x	REAL for slaruv DOUBLE PRECISION for dlaruv Array, DIMENSION (*n*). The generated random numbers.
seed	On exit, the seed is updated.

?larz

Applies an elementary reflector (as returned by ?tzrzf) to a general matrix.

Syntax

```
call slarz( side, m, n, l, v, incv, tau, c, ldc, work )
call dlarz( side, m, n, l, v, incv, tau, c, ldc, work )
call clarz( side, m, n, l, v, incv, tau, c, ldc, work )
call zlarz( side, m, n, l, v, incv, tau, c, ldc, work )
```

Include Files

- mkl.fi

Description

The routine ?larz applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right. *H* is represented in the forms

$H = I - tau * v * v^T$ for real flavors and $H = I - tau * v * v^H$ for complex flavors,

where *tau* is a real/complex scalar and v is a real/complex vector, respectively.

If $tau = 0$, then *H* is taken to be the unit matrix.

For complex flavors, to apply H^H (the conjugate transpose of *H*), supply conjg(*tau*) instead of *tau*.

H is a product of *k* elementary reflectors as returned by ?tzrzf.

Input Parameters

side	CHARACTER*1. If *side* = 'L': form *H***C* If *side* = 'R': form *C***H*
m	INTEGER. The number of rows of the matrix *C*.
n	INTEGER. The number of columns of the matrix *C*.
l	INTEGER. The number of entries of the vector *v* containing the meaningful part of the Householder vectors. If *side* = 'L', $m \geq l \geq 0$, if *side* = 'R', $n \geq l \geq 0$.

v
REAL for slarz
DOUBLE PRECISION for dlarz
COMPLEX for clarz
DOUBLE COMPLEX for zlarz
Array, DIMENSION (1+(*l*-1)*abs(*incv*)).
The vector *v* in the representation of *H* as returned by ?tzrzf.
v is not used if *tau* = 0.

incv
INTEGER. The increment between elements of *v*.
incv≠ 0.

tau
REAL for slarz
DOUBLE PRECISION for dlarz
COMPLEX for clarz
DOUBLE COMPLEX for zlarz
The value *tau* in the representation of *H*.

c
REAL for slarz
DOUBLE PRECISION for dlarz
COMPLEX for clarz
DOUBLE COMPLEX for zlarz
Array, DIMENSION (*ldc*,*n*).
On entry, the *m*-by-*n* matrix *C*.

ldc
INTEGER. The leading dimension of the array *c*.
ldc≥ max(1,*m*).

work
REAL for slarz
DOUBLE PRECISION for dlarz
COMPLEX for clarz
DOUBLE COMPLEX for zlarz
Workspace array, DIMENSION
(*n*) if *side* = 'L' or
(*m*) if *side* = 'R'.

Output Parameters

c
On exit, *C* is overwritten by the matrix *H***C* if *side* = 'L', or *C***H* if *side* = 'R'.

?larzb

Applies a block reflector or its transpose/conjugate-transpose to a general matrix.

Syntax

```
call slarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work, ldwork )
call dlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work, ldwork )
call clarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work, ldwork )
call zlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work, ldwork )
```

Include Files

- mkl.fi

Description

The routine applies a real/complex block reflector H or its transpose H^T (or the conjugate transpose H^H for complex flavors) to a real/complex distributed m-by-n matrix C from the left or the right. Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

side

CHARACTER*1.

If *side* = 'L': apply H or H^T/H^H from the left

If *side* = 'R': apply H or H^T/H^H from the right

trans

CHARACTER*1.

If *trans* = 'N': apply H (No transpose)

If *trans*='C': apply H^H (conjugate transpose)

If *trans*='T': apply H^T (transpose transpose)

direct

CHARACTER*1.

Indicates how H is formed from a product of elementary reflectors

= 'F': $H = H(1)*H(2)*...*H(k)$ (forward, not supported)

= 'B': $H = H(k)*...*H(2)*H(1)$ (backward)

storev

CHARACTER*1.

Indicates how the vectors which define the elementary reflectors are stored:

= 'C': Column-wise (not supported)

= 'R': Row-wise.

m

INTEGER. The number of rows of the matrix C.

n INTEGER. The number of columns of the matrix *C*.

k INTEGER. The order of the matrix *T* (equal to the number of elementary reflectors whose product defines the block reflector).

l INTEGER. The number of columns of the matrix *V* containing the meaningful part of the Householder reflectors.

If *side* = 'L', *m*≥*l*≥ 0, if *side* = 'R', *n*≥*l*≥ 0.

v REAL for slarzb

DOUBLE PRECISION for dlarzb

COMPLEX for clarzb

DOUBLE COMPLEX for zlarzb

Array, DIMENSION (*ldv*, nv).

If *storev* = 'C', nv = *k*;

if *storev* = 'R', nv = *l*.

ldv INTEGER. The leading dimension of the array *v*.

If *storev* = 'C', *ldv*≥*l*; if *storev* = 'R', *ldv*≥*k*.

t REAL for slarzb

DOUBLE PRECISION for dlarzb

COMPLEX for clarzb

DOUBLE COMPLEX for zlarzb

Array, DIMENSION (*ldt*,*k*). The triangular *k*-by-*k* matrix *T* in the representation of the block reflector.

ldt INTEGER. The leading dimension of the array *t*.

ldt≥*k*.

c REAL for slarzb

DOUBLE PRECISION for dlarzb

COMPLEX for clarzb

DOUBLE COMPLEX for zlarzb

Array, DIMENSION (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*.

ldc≥ max(1,*m*).

work REAL for slarzb

DOUBLE PRECISION for dlarzb

COMPLEX for clarzb

DOUBLE COMPLEX for zlarzb

Workspace array, DIMENSION (*ldwork*, *k*).

ldwork INTEGER. The leading dimension of the array *work*.

If *side* = 'L', *ldwork* ≥ max(1, *n*);

if *side* = 'R', *ldwork* ≥ max(1, *m*).

Output Parameters

c On exit, *C* is overwritten by $H*C$, or H^T/H^H*C, or $C*H$, or $C*H^T/H^H$.

?larzt

*Forms the triangular factor T of a block reflector H = I - V*T*V^H.*

Syntax

```
call slarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Include Files

- mkl.fi

Description

The routine forms the triangular factor *T* of a real/complex block reflector *H* of order > *n*, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', $H = H(1)*H(2)*...*H(k)$, and *T* is upper triangular.

If *direct* = 'B', $H = H(k)*...*H(2)*H(1)$, and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th column of the array *v*, and $H = I-V*T*V^T$ (for real flavors) or $H = I-V*T*V^H$ (for complex flavors).

If *storev* = 'R', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th row of the array *v*, and $H = I-V^T*T*V$ (for real flavors) or $H = I-V^H*T*V$ (for complex flavors).

Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

direct CHARACTER*1.

Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

If *direct* = 'F': $H = H(1)*H(2)*...*H(k)$ (forward, not supported)

If *direct* = 'B': $H = H(k)*...*H(2)*H(1)$ (backward)

storev CHARACTER*1.

Specifies how the vectors which define the elementary reflectors are stored (see also *Application Notes* below):

If *storev* = 'C': column-wise (not supported)

If *storev* = 'R': row-wise

n INTEGER. The order of the block reflector *H*. *n* ≥ 0.

k INTEGER. The order of the triangular factor *T* (equal to the number of elementary reflectors). *k* ≥ 1.

v REAL for slarzt

DOUBLE PRECISION for dlarzt

COMPLEX for clarzt

DOUBLE COMPLEX for zlarzt

Array, DIMENSION

(*ldv*, *k*) if *storev* = 'C'

(*ldv*, *n*) if *storev* = 'R' The matrix *V*.

ldv INTEGER. The leading dimension of the array *v*.

If *storev* = 'C', *ldv* ≥ max(1,*n*);

if *storev* = 'R', *ldv* ≥ *k*.

tau REAL for slarzt

DOUBLE PRECISION for dlarzt

COMPLEX for clarzt

DOUBLE COMPLEX for zlarzt

Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*).

ldt INTEGER. The leading dimension of the output array *t*.

ldt ≥ *k*.

Output Parameters

t REAL for slarzt

DOUBLE PRECISION for dlarzt

COMPLEX for clarzt

DOUBLE COMPLEX for zlarzt

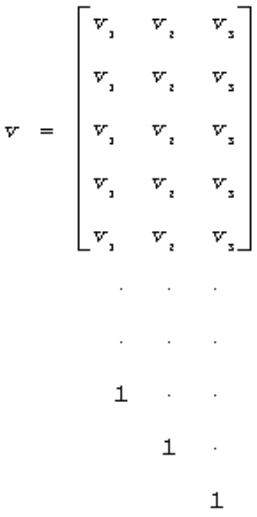
Array, DIMENSION (*ldt*,*k*). The *k*-by-*k* triangular factor *T* of the block reflector. If *direct* = 'F', *T* is upper triangular; if *direct* = 'B', *T* is lower triangular. The rest of the array is not used.

v The matrix *V*. See *Application Notes* below.

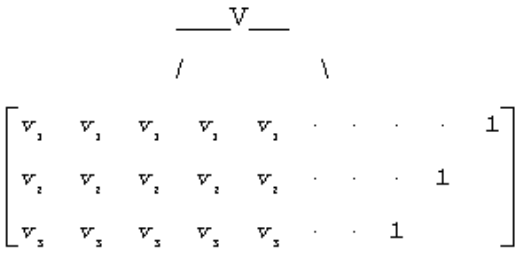
Application Notes

The shape of the matrix *V* and the storage of the vectors which define the *H*(i) is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

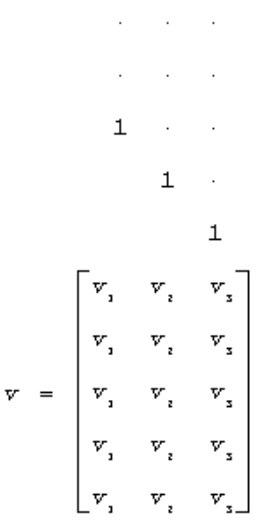
direct = 'F' and *storev* = 'C':



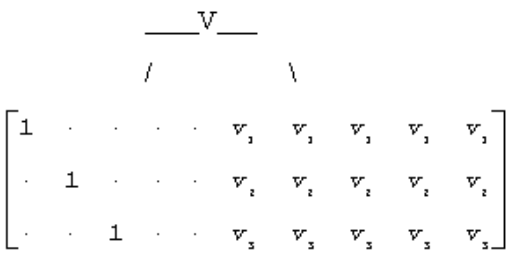
direct = 'F' and *storev* = 'R':



direct = 'B' and *storev* = 'C':



direct = 'B' and *storev* = 'R':



?las2

Computes singular values of a 2-by-2 triangular matrix.

Syntax

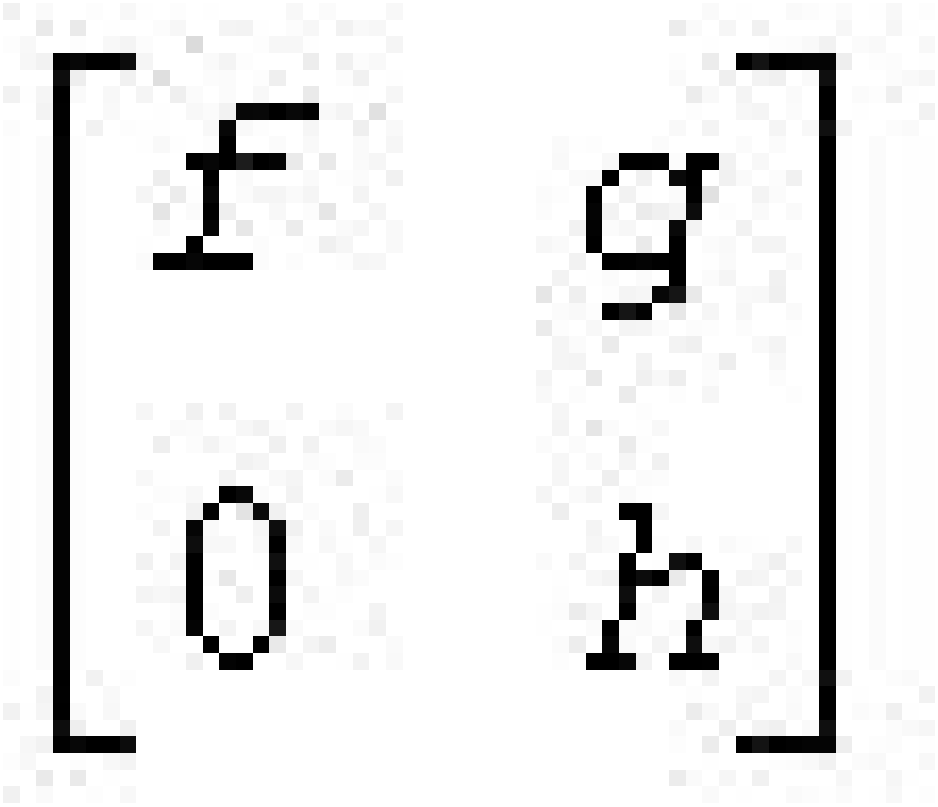
```
call slas2( f, g, h, ssmin, ssmax )
call dlas2( f, g, h, ssmin, ssmax )
```

Include Files

- mkl.fi

Description

The routine ?las2 computes the singular values of the 2-by-2 matrix



On return, *ssmin* is the smaller singular value and *SSMAX* is the larger singular value.

Input Parameters

f, *g*, *h*

REAL for slas2

DOUBLE PRECISION for dlas2

The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

ssmin, *ssmax*

REAL for slas2

DOUBLE PRECISION for dlas2

The smaller and the larger singular values, respectively.

Application Notes

Barring over/underflow, all output quantities are correct to within a few units in the last place (*ulps*), even in the absence of a guard digit in addition/subtraction. In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows, or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?lascl

Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from}.

Syntax

```
call slascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call dlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call clascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call zlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine ?lascl multiplies the *m*-by-*n* real/complex matrix *A* by the real scalar c_{to}/c_{from}. The operation is performed without over/underflow as long as the final result $c_{to}*A(i,j)/c_{from}$ does not over/underflow.

type specifies that *A* may be full, upper triangular, lower triangular, upper Hessenberg, or banded.

Input Parameters

type

CHARACTER*1. This parameter specifies the storage type of the input matrix.

= 'G': *A* is a full matrix.

= 'L': *A* is a lower triangular matrix.

= 'U': *A* is an upper triangular matrix.

= 'H': *A* is an upper Hessenberg matrix.

= 'B': *A* is a symmetric band matrix with lower bandwidth *kl* and upper bandwidth *ku* and with the only the lower half stored

= 'Q': *A* is a symmetric band matrix with lower bandwidth *kl* and upper bandwidth *ku* and with the only the upper half stored.

= 'Z': *A* is a band matrix with lower bandwidth *kl* and upper bandwidth *ku*. See description of the ?gbtrf function for storage details.

kl

INTEGER. The lower bandwidth of *A*. Referenced only if *type* = 'B', 'Q' or 'Z'.

ku

INTEGER. The upper bandwidth of *A*. Referenced only if *type* = 'B', 'Q' or 'Z'.

cfrom, *cto*

REAL for slascl/clascl

DOUBLE PRECISION for dlascl/zlascl

The matrix *A* is multiplied by *cto*/*cfrom*. A(*i*,*j*) is computed without over/underflow if the final result *cto**A(*i*,*j*)/*cfrom* can be represented without over/underflow. *cfrom* must be nonzero.

m	INTEGER. The number of rows of the matrix *A*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *A*. *n* ≥ 0.
a	REAL for slascl DOUBLE PRECISION for dlascl COMPLEX for clascl DOUBLE COMPLEX for zlascl Array, size (*lda*, *n*). The matrix to be multiplied by *cto*/*cfrom*. See *type* for the storage type.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*m*).

Output Parameters

a	The multiplied matrix *A*.
info	INTEGER. If *info* = 0 - successful exit If *info* = -*i* < 0, the *i*-th argument had an illegal value.

See Also

?gbtrf

?lasd0

Computes the singular values of a real upper bidiagonal n-by-m matrix B with diagonal d and off-diagonal e. Used by ?bdsdc.

Syntax

```
call slasd0( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info )
call dlasd0( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info )
```

Include Files

- mkl.fi

Description

Using a divide and conquer approach, the routine ?lasd0 computes the singular value decomposition (SVD) of a real upper bidiagonal *n*-by-*m* matrix *B* with diagonal *d* and offdiagonal *e*, where *m* = *n* + *sqre*.

The algorithm computes orthogonal matrices *U* and *VT* such that *B* = *U*S*VT*. The singular values *S* are overwritten on *d*.

The related subroutine ?lasda computes only the singular values, and optionally, the singular vectors in compact form.

Input Parameters

n — INTEGER. On entry, the row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array *d*.

sqre — INTEGER. Specifies the column dimension of the bidiagonal matrix.

If *sqre* = 0: the bidiagonal matrix has column dimension *m* = *n*.

If *sqre* = 1: the bidiagonal matrix has column dimension *m* = *n*+1.

d — REAL for slasd0

DOUBLE PRECISION for dlasd0

Array, DIMENSION (*n*). On entry, *d* contains the main diagonal of the bidiagonal matrix.

e — REAL for slasd0

DOUBLE PRECISION for dlasd0

Array, DIMENSION (*m*-1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.

ldu — INTEGER. On entry, leading dimension of the output array *u*.

ldvt — INTEGER. On entry, leading dimension of the output array *vt*.

smlsiz — INTEGER. On entry, maximum size of the subproblems at the bottom of the computation tree.

iwork — INTEGER.

Workspace array, dimension must be at least (8*n*).

work — REAL for slasd0

DOUBLE PRECISION for dlasd0

Workspace array, dimension must be at least $(3m^2+2m)$.

Output Parameters

d — On exit *d*, If *info* = 0, contains singular values of the bidiagonal matrix.

u — REAL for slasd0

DOUBLE PRECISION for dlasd0

Array, DIMENSION at least (*ldq*, *n*). On exit, *u* contains the left singular vectors.

vt — REAL for slasd0

DOUBLE PRECISION for dlasd0

Array, DIMENSION at least (*ldvt*, *m*). On exit, vt^T contains the right singular vectors.

info — INTEGER.

If *info* = 0: successful exit.

If *info* = -*i* < 0, the *i*-th argument had an illegal value.

If *info* = 1, a singular value did not converge.

?lasd1

Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by ?bdsdc.

Syntax

```
call slasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork, work, info )
call dlasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork, work, info )
```

Include Files

- mkl.fi

Description

The routine computes the SVD of an upper bidiagonal *n*-by-*m* matrix *B*, where *n* = *nl* + *nr* + 1 and *m* = *n* + *sqre*.

The routine ?lasd1 is called from ?lasd0.

A related subroutine ?lasd7 handles the case in which the singular values (and the singular vectors in factored form) are desired.

?lasd1 computes the SVD as follows:

$$VT = U(in)^* \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1^T & a & Z2^T & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} ^*VT(in)$$

= *U*(*out*)*(*D*(*out*) 0)**VT*(*out*)

whereZ^T = ($Z1^T a Z2^T b$) = $u^T * VT^T$, and *u* is a vector of dimension *m* with *alpha* and *beta* in the *nl*+1 and *nl* +2-th entries and zeros elsewhere; and the entry *b* is empty if *sqre* = 0.

The left singular vectors of the original matrix are stored in *u*, and the transpose of the right singular vectors are stored in *vt*, and the singular values are in *d*. The algorithm consists of three stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or when there are zeros in the *Z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?lasd2.
2. The second stage consists of calculating the updated singular values. This is done by finding the square roots of the roots of the secular equation via the routine ?lasd4 (as called by ?lasd3). This routine also calculates the singular vectors of the current problem.
3. The final stage consists of computing the updated singular vectors directly using the updated singular values. The singular vectors for the current problem are multiplied with the singular vectors from the overall problem.

Input Parameters

nl

INTEGER. The row dimension of the upper block.

nl ≥ 1.

nr

INTEGER. The row dimension of the lower block.

nr ≥ 1.

sqre

INTEGER.

If *sqre* = 0: the lower block is an *nr*-by-*nr* square matrix.

If *sqre* = 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has row dimension *n* = *nl* + *nr* + 1, and column dimension *m* = *n* + *sqre*.

d

REAL for slasd1

DOUBLE PRECISION for dlasd1

Array, DIMENSION (*nl*+*nr*+1). *n* = *nl*+*nr*+1. On entry *d*(1:*nl*,1:*nl*) contains the singular values of the upper block; and *d*(*nl*+2:*n*) contains the singular values of the lower block.

alpha

REAL for slasd1

DOUBLE PRECISION for dlasd1

Contains the diagonal element associated with the added row.

beta

REAL for slasd1

DOUBLE PRECISION for dlasd1

Contains the off-diagonal element associated with the added row.

u

REAL for slasd1

DOUBLE PRECISION for dlasd1

Array, DIMENSION (*ldu*, *n*). On entry *u*(1:*nl*, 1:*nl*) contains the left singular vectors of the upper block; *u*(*nl*+2:*n*, *nl*+2:*n*) contains the left singular vectors of the lower block.

ldu

INTEGER. The leading dimension of the array *U*.

ldu ≥ max(1, *n*).

vt

REAL for slasd1

DOUBLE PRECISION for dlasd1

Array, DIMENSION (*ldvt*, *m*), where *m* = *n* + *sqre*.

On entry $vt(1:nl+1, 1:nl+1)^T$ contains the right singular vectors of the upper block; $vt(nl+2:m, nl+2:m)^T$ contains the right singular vectors of the lower block.

ldvt

INTEGER. The leading dimension of the array *vt*.

ldvt ≥ max(1, *M*).

iwork

INTEGER.

Workspace array, DIMENSION (4*n*).

work

REAL for slasd1

DOUBLE PRECISION for dlasd1

Workspace array, DIMENSION ($3m_2 + 2m$).

Output Parameters

d

On exit *d*(1:*n*) contains the singular values of the modified matrix.

alpha

On exit, the diagonal element associated with the added row deflated by max(abs(*alpha*), abs(*beta*), abs(D(I))), I = 1,n.

beta

On exit, the off-diagonal element associated with the added row deflated by max(abs(*alpha*), abs(*beta*), abs(D(I))), I = 1,n.

u

On exit *u* contains the left singular vectors of the bidiagonal matrix.

vt

On exit vt^T contains the right singular vectors of the bidiagonal matrix.

idxq

INTEGER.

Array, DIMENSION (*n*). Contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, *d*(*idxq*(*i* = 1, *n*)) will be in ascending order.

info

INTEGER.

If *info* = 0: successful exit.

If *info* = -*i* < 0, the *i*-th argument had an illegal value.

If *info* = 1, a singular value did not converge.

?lasd2

Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.

Syntax

call slasd2(*nl*, *nr*, *sqre*, *k*, *d*, *z*, *alpha*, *beta*, *u*, *ldu*, *vt*, *ldvt*, *dsigma*, *u2*, *ldu2*, *vt2*, *ldvt2*, *idxp*, *idx*, *idxp*, *idxq*, *coltyp*, *info*)

call dlasd2(*nl*, *nr*, *sqre*, *k*, *d*, *z*, *alpha*, *beta*, *u*, *ldu*, *vt*, *ldvt*, *dsigma*, *u2*, *ldu2*, *vt2*, *ldvt2*, *idxp*, *idx*, *idxp*, *idxq*, *coltyp*, *info*)

Include Files

- mkl.fi

Description

The routine ?lasd2 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the *Z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

The routine ?lasd2 is called from ?lasd1.

Input Parameters

nl — INTEGER. The row dimension of the upper block.

nl≥ 1.

nr — INTEGER. The row dimension of the lower block.

nr≥ 1.

sqre — INTEGER.

If *sqre* = 0): the lower block is an *nr*-by-*nr* square matrix

If *sqre* = 1): the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has *n* = *nl* + *nr* + 1 rows and *m* = *n* + *sqre*≥*n* columns.

d — REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION (*n*). On entry *d* contains the singular values of the two submatrices to be combined.

alpha — REAL for slasd2

DOUBLE PRECISION for dlasd2

Contains the diagonal element associated with the added row.

beta — REAL for slasd2

DOUBLE PRECISION for dlasd2

Contains the off-diagonal element associated with the added row.

u — REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION (*ldu*, *n*). On entry *u* contains the left singular vectors of two submatrices in the two square blocks with corners at (1,1), (*nl*, *nl*), and (*nl*+2, *nl*+2), (*n*,*n*).

ldu — INTEGER. The leading dimension of the array *u*.

ldu≥*n*.

ldu2 — INTEGER. The leading dimension of the output array *u2*. *ldu2*≥*n*.

vt — REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION (*ldvt*, *m*). On entry, vt^T contains the right singular vectors of two submatrices in the two square blocks with corners at (1,1), (*nl*+1, *nl*+1), and (*nl*+2, *nl*+2), (*m*, *m*).

ldvt — INTEGER. The leading dimension of the array *vt*. *ldvt*≥*m*.

ldvt2 — INTEGER. The leading dimension of the output array *vt2*. *ldvt2*≥*m*.

idxp — INTEGER.

Workspace array, DIMENSION (*n*). This will contain the permutation used to place deflated values of *D* at the end of the array. On output *idxp*(2:*k*) points to the nondeflated *d*-values and *idxp*(*k*+1:*n*) points to the deflated singular values.

idx

INTEGER.

Workspace array, DIMENSION (*n*). This will contain the permutation used to sort the contents of *d* into ascending order.

coltyp

INTEGER.

Workspace array, DIMENSION (*n*). As workspace, this array contains a label that indicates which of the following types a column in the *u2* matrix or a row in the *vt2* matrix is:

1 : non-zero in the upper half only

2 : non-zero in the lower half only

3 : dense

4 : deflated.

idxq

INTEGER. Array, DIMENSION (*n*). This parameter contains the permutation that separately sorts the two sub-problems in *D* in the ascending order. Note that entries in the first half of this permutation must first be moved one position backwards and entries in the second half must have *nl*+1 added to their values.

Output Parameters

k

INTEGER. Contains the dimension of the non-deflated matrix, This is the order of the related secular equation. $1 \leq k \leq n$.

d

On exit *D* contains the trailing (*n*-*k*) updated singular values (those which were deflated) sorted into increasing order.

u

On exit *u* contains the trailing (*n*-*k*) updated left singular vectors (those which were deflated) in its last *n*-*k* columns.

z

REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION (*n*). On exit, *z* contains the updating row vector in the secular equation.

dsigma

REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION (*n*). Contains a copy of the diagonal elements (*k*-1 singular values and one zero) in the secular equation.

u2

REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION (*ldu2*, *n*). Contains a copy of the first *k*-1 left singular vectors which will be used by ?lasd3 in a matrix multiply (?gemm) to solve for the new left singular vectors. *u2* is arranged into four blocks. The first

block contains a column with 1 at *nl*+1 and zero everywhere else; the second block contains non-zero entries only at and above *nl*; the third contains non-zero entries only below *nl*+1; and the fourth is dense.

vt — On exit, vt^T contains the trailing (*n*-*k*) updated right singular vectors (those which were deflated) in its last *n*-*k* columns. In case *sqre* =1, the last row of *vt* spans the right null space.

vt2 — REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION (*ldvt2*, *n*). $vt2^T$ contains a copy of the first *k* right singular vectors which will be used by ?lasd3 in a matrix multiply (?gemm) to solve for the new right singular vectors. *vt2* is arranged into three blocks. The first block contains a row that corresponds to the special 0 diagonal element in *sigma*; the second block contains non-zeros only at and before *nl* +1; the third block contains non-zeros only at and after *nl* +2.

idxc — INTEGER. Array, DIMENSION (*n*). This will contain the permutation used to arrange the columns of the deflated *u* matrix into three groups: the first group contains non-zero entries only at and above *nl*, the second contains non-zero entries only below *nl*+2, and the third is dense.

coltyp — On exit, it is an array of dimension 4, with *coltyp*(*i*) being the dimension of the *i*-th type columns.

info — INTEGER.

If *info* = 0): successful exit

If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lasd3

Finds all square roots of the roots of the secular equation, as defined by the values in D and Z, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.

Syntax

```
call slasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt, vt2, ldvt2,
idxc, ctot, z, info )
```

```
call dlasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt, vt2, ldvt2,
idxc, ctot, z, info )
```

Include Files

- mkl.fi

Description

The routine ?lasd3 finds all the square roots of the roots of the secular equation, as defined by the values in *D* and *Z*.

It makes the appropriate calls to ?lasd4 and then updates the singular vectors by matrix multiplication.

The routine ?lasd3 is called from ?lasd1.

Input Parameters

nl

INTEGER. The row dimension of the upper block.

nl≥ 1.

nr

INTEGER. The row dimension of the lower block.

nr≥ 1.

sqre

INTEGER.

If *sqre* = 0): the lower block is an *nr*-by-*nr* square matrix.

If *sqre* = 1): the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has *n* = *nl* + *nr* + 1 rows and *m* = *n* + *sqre*≥*n* columns.

k

INTEGER.The size of the secular equation, 1 ≤ *k* ≤ *n*.

q

REAL for slasd3

DOUBLE PRECISION for dlasd3

Workspace array, DIMENSION at least (*ldq*, *k*).

ldq

INTEGER. The leading dimension of the array *Q*.

ldq≥*k*.

dsigma

REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (*k*). The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

ldu

INTEGER. The leading dimension of the array *u*.

ldu≥*n*.

u2

REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (*ldu2*, *n*).

The first *k* columns of this matrix contain the non-deflated left singular vectors for the split problem.

ldu2

INTEGER. The leading dimension of the array *u2*.

ldu2≥*n*.

ldvt

INTEGER. The leading dimension of the array *vt*.

ldvt≥*n*.

vt2

REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (*ldvt2*, *n*).

The first *k* columns of *vt2*' contain the non-deflated right singular vectors for the split problem.

ldvt2 INTEGER. The leading dimension of the array *vt2*.

ldvt2≥*n*.

idxc INTEGER. Array, DIMENSION (*n*).

The permutation used to arrange the columns of u (and rows of *vt*) into three groups: the first group contains non-zero entries only at and above (or before) *nl* +1; the second contains non-zero entries only at and below (or after) *nl*+2; and the third is dense. The first column of *u* and the row of *vt* are treated separately, however. The rows of the singular vectors found by ?lasd4 must be likewise permuted before the matrix multiplies can take place.

ctot INTEGER. Array, DIMENSION (4). A count of the total number of the various types of columns in *u* (or rows in *vt*), as described in *idxc*.

The fourth column type is any column which has been deflated.

z REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (*k*). The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

Output Parameters

d REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (*k*). On exit the square roots of the roots of the secular equation, in ascending order.

u REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (*ldu*, *n*).

The last *n* - *k* columns of this matrix contain the deflated left singular vectors.

vt REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (*ldvt*, *m*).

The last *m* – *k* columns of *vt*' contain the deflated right singular vectors.

vt2 Destroyed on exit.

z Destroyed on exit.

info INTEGER.

If *info* = 0): successful exit.

If *info* = -*i* < 0, the *i*-th argument had an illegal value.

If *info* = 1, an singular value did not converge.

Application Notes

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

?lasd4

Computes the square root of the i-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc*.*

Syntax

```
call slasd4( n, i, d, z, delta, rho, sigma, work, info)
call dlasd4( n, i, d, z, delta, rho, sigma, work, info )
```

Include Files

- mkl.fi

Description

The routine computes the square root of the *i*-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix whose entries are given as the squares of the corresponding entries in the array *d*, and that $0 \le d(i) < d(j)$ for $i < j$ and that $rho > 0$. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

$diag(d)*diag(d) + rho*Z*Z^T$,

where the Euclidean norm of *Z* is equal to 1.The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

n

INTEGER. The length of all arrays.

i

INTEGER.

The index of the eigenvalue to be computed. $1 \le i \le n$.

d

REAL for slasd4

DOUBLE PRECISION for dlasd4

Array, DIMENSION (*n*).

The original eigenvalues. They must be in order, $0 \le d(i) < d(j)$ for $i < j$.

z

REAL for slasd4

DOUBLE PRECISION for dlasd4

Array, DIMENSION (*n*).

The components of the updating vector.

rho

REAL for slasd4

DOUBLE PRECISION for dlasd4

The scalar in the symmetric updating formula.

work

REAL for slasd4

DOUBLE PRECISION for dlasd4

Workspace array, DIMENSION (*n*).

If *n*≠ 1, *work* contains (*d*(*j*) + *sigma_i*) in its *j*-th component.

If *n* = 1, then *work*(1) = 1.

Output Parameters

delta

REAL for slasd4

DOUBLE PRECISION for dlasd4

Array, DIMENSION (*n*).

If *n*≠ 1, *delta* contains (*d*(*j*) – *sigma_i*) in its *j*-th component.

If *n* = 1, then *delta* (1) = 1. The vector *delta* contains the information necessary to construct the (singular) eigenvectors.

sigma

REAL for slasd4

DOUBLE PRECISION for dlasd4

The computed *sigma_i*, the *i*-th updated eigenvalue.

info

INTEGER.

= 0: successful exit

> 0: If *info* = 1, the updating process failed.

?lasd5

Computes the square root of the i-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix.Used by ?bdsdc.

Syntax

```
call slasd5( i, d, z, delta, rho, dsigma, work )
call dlasd5( i, d, z, delta, rho, dsigma, work )
```

Include Files

- mkl.fi

Description

The routine computes the square root of the *i*-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix diag(*d*)*diag(*d*)+*rho***Z***Z^T

The diagonal entries in the array *d* must satisfy `0 ≤ d(i) < d(j)` for `i<i`, *rho* mustbe greater than 0, and that the Euclidean norm of the vector *Z* is equal to 1.

Input Parameters

i — `INTEGER`.The index of the eigenvalue to be computed. `i = 1` or `i = 2`.

d — `REAL` for `slasd5`

`DOUBLE PRECISION` for `dlasd5`

Array, `dimension` (2).

The original eigenvalues, `0 ≤ d(1) < d(2)`.

z — `REAL` for `slasd5`

`DOUBLE PRECISION` for `dlasd5`

Array, `dimension` (2).

The components of the updating vector.

rho — `REAL` for `slasd5`

`DOUBLE PRECISION` for `dlasd5`

The scalar in the symmetric updating formula.

work — `REAL` for `slasd5`

`DOUBLE PRECISION` for `dlasd5`

Workspace array, `dimension` (2). Contains (`d(j) + sigma_i`) in its *j*-th component.

Output Parameters

delta — `REAL` for `slasd5`

`DOUBLE PRECISION` for `dlasd5`

Array, `dimension` (2).

Contains (`d(j) - sigma_i`) in its *j*-th component. The vector *delta* contains the information necessary to construct the eigenvectors.

dsigma — `REAL` for `slasd5`

`DOUBLE PRECISION` for `dlasd5`

The computed *sigma_i*, the *i*-th updated eigenvalue.

?lasd6

Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by `?bdsdc`.

Syntax

```
call slasd6( icompq, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr, givcol,
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork, info )
```

```
call dlasd6( icompq, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr, givcol,
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork, info )
```

Include Files

- mkl.fi

Description

The routine ?lasd6 computes the *SVD* of an updated upper bidiagonal matrix *B* obtained by merging two smaller ones by appending a row. This routine is used only for the problem which requires all singular values and optionally singular vector matrices in factored form. *B* is an *n*-by-*m* matrix with $n = nl + nr + 1$ and $m = n + sqre$. A related subroutine, ?lasd1, handles the case in which all singular values and singular vectors of the bidiagonal matrix are desired. ?lasd6 computes the *SVD* as follows:

$$B = U(in)*\begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1' & a & Z2' & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix}*VT(in)$$

$= U(\text{out})*(D(\text{out})*VT(out)$

where $Z' = (Z1'\ aZ2'\ b) = u'*VT'$, and *u* is a vector of dimension *m* with *alpha* and *beta* in the *nl*+1 and *nl*+2-th entries and zeros elsewhere; and the entry b is empty if $sqre = 0$.

The singular values of *B* can be computed using *D1*, *D2*, the first components of all the right singular vectors of the lower block, and the last components of all the right singular vectors of the upper block. These components are stored and updated in *vf* and *vl*, respectively, in ?lasd6. Hence *U* and *VT* are not explicitly referenced.

The singular values are stored in *D*. The algorithm consists of two stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or if there is a zero in the *Z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?lasd7.
2. The second stage consists of calculating the updated singular values. This is done by finding the roots of the secular equation via the routine ?lasd4 (as called by ?lasd8). This routine also updates *vf* and *vl* and computes the distances between the updated singular values and the old singular values. ?lasd6 is called from ?lasda.

Input Parameters

icompq — INTEGER. Specifies whether singular vectors are to be computed in factored form:

= 0: Compute singular values only

= 1: Compute singular vectors in factored form as well.

nl — INTEGER. The row dimension of the upper block.

$nl \geq 1$.

nr — INTEGER. The row dimension of the lower block.

	nr≥ 1.
sqre	INTEGER. = 0: the lower block is an *nr*-by-*nr* square matrix. = 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has row dimension *n*=*nl*+*nr*+1, and column dimension *m* = *n* + *sqre*.
d	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (*nl*+*nr*+1). On entry *d*(1:*nl*,1:*nl*) contains the singular values of the upper block, and *d*(*nl*+2:*n*) contains the singular values of the lower block.
vf	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (*m*). On entry, *vf*(1:*nl*+1) contains the first components of all right singular vectors of the upper block; and *vf*(*nl*+2:*m*) contains the first components of all right singular vectors of the lower block.
vl	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (*m*). On entry, *vl*(1:*nl*+1) contains the last components of all right singular vectors of the upper block; and *vl*(*nl*+2:*m*) contains the last components of all right singular vectors of the lower block.
alpha	REAL for slasd6 DOUBLE PRECISION for dlasd6 Contains the diagonal element associated with the added row.
beta	REAL for slasd6 DOUBLE PRECISION for dlasd6 Contains the off-diagonal element associated with the added row.
ldgcol	INTEGER.The leading dimension *of* the output array *givcol*, must be at least *n*.
ldgnum	INTEGER The leading dimension of the output arrays *givnum* and *poles*, must be at least *n*.
work	REAL for slasd6 DOUBLE PRECISION for dlasd6 Workspace array, dimension (4*m*).

iwork INTEGER

Workspace array, dimension (3*n*).

Output Parameters

d On exit *d*(1:*n*) contains the singular values of the modified matrix.

vf On exit, *vf* contains the first components of all right singular vectors of the bidiagonal matrix.

vl On exit, *vl* contains the last components of all right singular vectors of the bidiagonal matrix.

alpha On exit, the diagonal element associated with the added row deflated by max(abs(*alpha*), abs(*beta*), abs(D(I))), I = 1,n.

beta On exit, the off-diagonal element associated with the added row deflated by max(abs(*alpha*), abs(*beta*), abs(D(I))), I = 1,n.

idxq INTEGER .

Array, dimension (*n*). This contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, *d*(*idxq*(*i* = 1, *n*)) will be in ascending order.

perm INTEGER .

Array, dimension (*n*). The permutations (from deflation and sorting) to be applied to each block. Not referenced if *icompq* = 0.

givptr INTEGER. The number of Givens rotations which took place in this subproblem. Not referenced if *icompq* = 0.

givcol INTEGER .

Array, dimension (*ldgcol*, 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if *icompq* = 0.

givnum REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension (*ldgnum*, 2). Each number indicates the *C* or *S* value to be used in the corresponding Givens rotation. Not referenced if *icompq* = 0.

poles REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension (*ldgnum*, 2). On exit, *poles*(1,*) is an array containing the new singular values obtained from solving the secular equation, and *poles*(2,*) is an array containing the poles in the secular equation. Not referenced if *icompq* = 0.

difl REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension (*n*). On exit, *difl*(*i*) is the distance between *i*-th updated (undeflated) singular value and the *i*-th (undeflated) old singular value.

difr

REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension (*ldgnum*, 2) if *icompq* = 1 and dimension (*n*) if *icompq* = 0.

On exit, *difr*(*i*, 1) is the distance between *i*-th updated (undeflated) singular value and the *i*+1-th (undeflated) old singular value. If *icompq* = 1, *difr*(1: *k*, 2) is an array containing the normalizing factors for the right singular vector matrix.

See ?lasd8 for details on *difl* and *difr*.

z

REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension (*m*).

The first elements of this array contain the components of the deflation-adjusted updating row vector.

k

INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. 1 ≤ *k* ≤ *n*.

c

REAL for slasd6

DOUBLE PRECISION for dlasd6

c contains garbage if *sqre* =0 and the *C*-value of a Givens rotation related to the right null space if *sqre* = 1.

s

REAL for slasd6

DOUBLE PRECISION for dlasd6

s contains garbage if *sqre* =0 and the *S*-value of a Givens rotation related to the right null space if *sqre* = 1.

info

INTEGER .

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

> 0: if *info* = 1, an singular value did not converge

?lasd7

Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.

Syntax

call slasd7(*icompq*, *nl*, *nr*, *sqre*, *k*, *d*, *z*, *zw*, *vf*, *vfw*, *vl*, *vlw*, *alpha*, *beta*, *dsigma*, *idx*, *idxp*, *idxq*, *perm*, *givptr*, *givcol*, *ldgcol*, *givnum*, *ldgnum*, *c*, *s*, *info*)

call dlasd7(*icompq*, *nl*, *nr*, *sqre*, *k*, *d*, *z*, *zw*, *vf*, *vfw*, *vl*, *vlw*, *alpha*, *beta*, *dsigma*, *idx*, *idxp*, *idxq*, *perm*, *givptr*, *givcol*, *ldgcol*, *givnum*, *ldgnum*, *c*, *s*, *info*)

Include Files

- mkl.fi

Description

The routine ?lasd7 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the *Z* vector. For each such occurrence the order of the related secular equation problem is reduced by one. ?lasd7 is called from ?lasd6.

Input Parameters

icompq

INTEGER. Specifies whether singular vectors are to be computed in compact form, as follows:

= 0: Compute singular values only.

= 1: Compute singular vectors of upper bidiagonal matrix in compact form.

nl

INTEGER. The row dimension of the upper block.

nl ≥ 1.

nr

INTEGER. The row dimension of the lower block.

nr ≥ 1.

sqre

INTEGER.

= 0: the lower block is an *nr*-by-*nr* square matrix.

= 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has *n* = *nl* + *nr* + 1 rows and *m* = *n* + *sqre* ≥ *n* columns.

d

REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Array, DIMENSION (*n*). On entry *d* contains the singular values of the two submatrices to be combined.

zw

REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Array, DIMENSION (*m*).

Workspace for z.

vf

REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Array, DIMENSION (*m*). On entry, *vf*(1:*nl*+1) contains the first components of all right singular vectors of the upper block; and *vf*(*nl* +2:*m*) contains the first components of all right singular vectors of the lower block.

vfw

REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Array, DIMENSION (*m*).

Workspace for vf.

vl

REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Array, DIMENSION (*m*).

On entry, *vl*(1:*nl*+1) contains the last components of all right singular vectors of the upper block; and *vl*(*nl*+2:*m*) contains the last components of all right singular vectors of the lower block.

VLW

REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Array, DIMENSION (*m*).

Workspace for VL.

alpha

REAL for slasd7

DOUBLE PRECISION for dlasd7.

Contains the diagonal element associated with the added row.

beta

REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Contains the off-diagonal element associated with the added row.

idx

INTEGER.

Workspace array, DIMENSION (*n*). This will contain the permutation used to sort the contents of *d* into ascending order.

idxp

INTEGER.

Workspace array, DIMENSION (*n*). This will contain the permutation used to place deflated values of *d* at the end of the array.

idxq

INTEGER.

Array, DIMENSION (*n*).

This contains the permutation which separately sorts the two sub-problems in *d* into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have *nl*+1 added to their values.

ldgcol

INTEGER.The leading dimension of the output array *givcol*, must be at least *n*.

ldgnum

INTEGER. The leading dimension of the output array *givnum*, must be at least *n*.

Output Parameters

k

INTEGER. Contains the dimension of the non-deflated matrix, this is the order of the related secular equation.

$1 \leq k \leq n$.

d — On exit, *d* contains the trailing (*n*-*k*) updated singular values (those which were deflated) sorted into increasing order.

z — REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Array, DIMENSION (*m*).

On exit, *Z* contains the updating row vector in the secular equation.

vf — On exit, *vf* contains the first components of all right singular vectors of the bidiagonal matrix.

vl — On exit, *vl* contains the last components of all right singular vectors of the bidiagonal matrix.

dsigma — REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Array, DIMENSION (*n*). Contains a copy of the diagonal elements (*k*-1 singular values and one zero) in the secular equation.

idxp — On output, *idxp*(2: *k*) points to the nondeflated *d*-values and *idxp*(*k*+1:*n*) points to the deflated singular values.

perm — INTEGER.

Array, DIMENSION (*n*).

The permutations (from deflation and sorting) to be applied to each singular block. Not referenced if *icompq* = 0.

givptr — INTEGER.

The number of Givens rotations which took place in this subproblem. Not referenced if *icompq* = 0.

givcol — INTEGER.

Array, DIMENSION (*ldgcol*, 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if *icompq* = 0.

givnum — REAL for slasd7.

DOUBLE PRECISION for dlasd7.

Array, DIMENSION (*ldgnum*, 2). Each number indicates the *C* or *S* value to be used in the corresponding Givens rotation. Not referenced if *icompq* = 0.

c — REAL for slasd7.

DOUBLE PRECISION for dlasd7.

If *sqre* =0, then *c* contains garbage, and if *sqre* = 1, then *c* contains *C*-value of a Givens rotation related to the right null space.

S — REAL for slasd7.

DOUBLE PRECISION for dlasd7.

If *sqre* =0, then *s* contains garbage, and if *sqre* = 1, then *s* contains *S*-value of a Givens rotation related to the right null space.

info INTEGER.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?lasd8

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
call slasd8( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
call dlasd8( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
```

Include Files

- mkl.fi

Description

The routine ?lasd8 finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to ?lasd4, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd8 is called from ?lasd6.

Input Parameters

icompq INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine:

= 0: Compute singular values only.

= 1: Compute singular vectors in factored form as well.

k INTEGER. The number of terms in the rational function to be solved by ?lasd4. $k \geq 1$.

z REAL for slasd8

DOUBLE PRECISION for dlasd8.

Array, DIMENSION (*k*).

The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

vf REAL for slasd8

DOUBLE PRECISION for dlasd8.

Array, DIMENSION (*k*).

On entry, *vf* contains information passed through dbede8.

vl

REAL for slasd8

DOUBLE PRECISION for dlasd8.

Array, DIMENSION (*k*). On entry, *vl* contains information passed through dbede8.

lddifr

INTEGER. The leading dimension of the output array *difr*, must be at least *k*.

dsigma

REAL for slasd8

DOUBLE PRECISION for dlasd8.

Array, DIMENSION (*k*).

The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

work

REAL for slasd8

DOUBLE PRECISION for dlasd8.

Workspace array, DIMENSION at least (3*k*).

Output Parameters

d

REAL for slasd8

DOUBLE PRECISION for dlasd8.

Array, DIMENSION (*k*).

On output, *D* contains the updated singular values.

z

Updated on exit.

vf

On exit, *vf* contains the first *k* components of the first components of all right singular vectors of the bidiagonal matrix.

vl

On exit, *vl* contains the first *k* components of the last components of all right singular vectors of the bidiagonal matrix.

difl

REAL for slasd8

DOUBLE PRECISION for dlasd8.

Array, DIMENSION (*k*). On exit, *difl*(*i*) = *d*(*i*) - *dsigma*(*i*).

difr

REAL for slasd8

DOUBLE PRECISION for dlasd8.

Array,

DIMENSION (*lddifr*, 2) if *icompq* = 1 and

DIMENSION (*k*) if *icompq* = 0.

On exit, *difr*(i,1) = *d*(*i*) - *dsigma*(i+1), *difr*(*k*,1) is not defined and will not be referenced. If *icompq* = 1, *difr*(1:*k*,2) is an array containing the normalizing factors for the right singular vector matrix.

dsigma

The elements of this array may be very slightly altered in value.

info INTEGER.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

> 0: If *info* = 1, an singular value did not converge.

?lasd9

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
call slasd9( icompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
call dlasd9( icompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
```

Include Files

- mkl.fi

Description

The routine ?lasd9 finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to ?lasd4, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd9 is called from ?lasd7.

Input Parameters

icompq INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine:

If *icompq* = 0, compute singular values only;

If *icompq* = 1, compute singular vector matrices in factored form also.

k INTEGER. The number of terms in the rational function to be solved by slasd4. $k \geq 1$.

dsigma REAL for slasd9

DOUBLE PRECISION for dlasd9.

Array, DIMENSION(*k*).

The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

z REAL for slasd9

DOUBLE PRECISION for dlasd9.

Array, DIMENSION (*k*). The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

vf REAL for slasd9

DOUBLE PRECISION for dlasd9.

Array, DIMENSION(*k*). On entry, *vf* contains information passed through sbede8.

vl

REAL for slasd9

DOUBLE PRECISION for dlasd9.

Array, DIMENSION(*k*). On entry, *vl* contains information passed through sbede8.

work

REAL for slasd9

DOUBLE PRECISION for dlasd9.

Workspace array, DIMENSION at least (3*k*).

Output Parameters

d

REAL for slasd9

DOUBLE PRECISION for dlasd9.

Array, DIMENSION(*k*). *d*(*i*) contains the updated singular values.

vf

On exit, *vf* contains the first *k* components of the first components of all right singular vectors of the bidiagonal matrix.

vl

On exit, *vl* contains the first *k* components of the last components of all right singular vectors of the bidiagonal matrix.

difl

REAL for slasd9

DOUBLE PRECISION for dlasd9.

Array, DIMENSION (*k*).

On exit, *difl*(*i*) = *d*(*i*) - *dsigma*(*i*).

difr

REAL for slasd9

DOUBLE PRECISION for dlasd9.

Array,

DIMENSION (*ldu*, 2) if *icompq* =1 and

DIMENSION (*k*) if *icompq* = 0.

On exit, *difr*(*i*, 1) = *d*(*i*) - *dsigma*(*i*+1), *difr*(*k*, 1) is not defined and will not be referenced.

If *icompq* = 1, *difr*(1:*k*, 2) is an array containing the normalizing factors for the right singular vector matrix.

info

INTEGER.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

> 0: If *info* = 1, an singular value did not converge

?lasda

Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal d and off-diagonal e. Used by ?bdsdc.

Syntax

call slasda(*icompq*, *smlsiz*, *n*, *sqre*, *d*, *e*, *u*, *ldu*, *vt*, *k*, *difl*, *difr*, *z*, *poles*, *givptr*, *givcol*, *ldgcol*, *perm*, *givnum*, *c*, *s*, *work*, *iwork*, *info*)

call dlasda(*icompq*, *smlsiz*, *n*, *sqre*, *d*, *e*, *u*, *ldu*, *vt*, *k*, *difl*, *difr*, *z*, *poles*, *givptr*, *givcol*, *ldgcol*, *perm*, *givnum*, *c*, *s*, *work*, *iwork*, *info*)

Include Files

- mkl.fi

Description

Using a divide and conquer approach, ?lasda computes the singular value decomposition (*SVD*) of a real upper bidiagonal *n*-by-*m* matrix *B* with diagonal *d* and off-diagonal *e*, where *m = n + sqre*.

The algorithm computes the singular values in the *SVD B = U*S*VT*. The orthogonal matrices *U* and *VT* are optionally computed in compact form. A related subroutine ?lasd0 computes the singular values and the singular vectors in explicit form.

Input Parameters

icompq — INTEGER.

Specifies whether singular vectors are to be computed in compact form, as follows:

= 0: Compute singular values only.

= 1: Compute singular vectors of upper bidiagonal matrix in compact form.

smlsiz — INTEGER.

The maximum size of the subproblems at the bottom of the computation tree.

n — INTEGER. The row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array *d*.

sqre — INTEGER. Specifies the column dimension of the bidiagonal matrix.

If *sqre = 0*: the bidiagonal matrix has column dimension *m = n*

If *sqre = 1*: the bidiagonal matrix has column dimension *m = n + 1*.

d — REAL for slasda

DOUBLE PRECISION for dlasda.

Array, DIMENSION (*n*). On entry, *d* contains the main diagonal of the bidiagonal matrix.

e — REAL for slasda

DOUBLE PRECISION for dlasda.

Array, DIMENSION (*m* - 1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.

ldu — INTEGER. The leading dimension of arrays *u*, *vt*, *difl*, *difr*, *poles*, *givnum*, and *z*. $ldu \geq n$.

ldgcol — INTEGER. The leading dimension of arrays *givcol* and *perm*. $ldgcol \geq n$.

work — REAL for slasda

DOUBLE PRECISION for dlasda.

Workspace array, DIMENSION $(6n+(smlsiz+1)^2)$.

iwork — INTEGER.

Workspace array, *Dimension* must be at least (7*n*).

Output Parameters

d — On exit *d*, if $info = 0$, contains the singular values of the bidiagonal matrix.

u — REAL for slasda

DOUBLE PRECISION for dlasda.

Array, DIMENSION (*ldu*, *smlsiz*) if $icompq = 1$.

Not referenced if $icompq = 0$.

If $icompq = 1$, on exit, *u* contains the left singular vector matrices of all subproblems at the bottom level.

vt — REAL for slasda

DOUBLE PRECISION for dlasda.

Array, DIMENSION (*ldu*, *smlsiz*+1) if $icompq = 1$, and not referenced if $icompq = 0$. If $icompq = 1$, on exit, *vt*' contains the right singular vector matrices of all subproblems at the bottom level.

k — INTEGER.

Array, DIMENSION (*n*) if $icompq = 1$ and

DIMENSION (1) if $icompq = 0$.

If $icompq = 1$, on exit, *k*(*i*) is the dimension of the *i*-th secular equation on the computation tree.

difl — REAL for slasda

DOUBLE PRECISION for dlasda.

Array, DIMENSION (*ldu*, *nlvl*),

where $nlvl = \text{floor}(\log_2(n/smlsiz))$.

difr — REAL for slasda

DOUBLE PRECISION for dlasda.

Array,

DIMENSION (*ldu*, 2 *nlvl*) if *icompq* = 1 and

DIMENSION (*n*) if *icompq* = 0.

If *icompq* = 1, on exit, *difl*(1:*n*, *i*) and *difr*(1:*n*,2*i* -1) record distances between singular values on the *i*-th level and singular values on the (*i* -1)-th level, and *difr*(1:*n*, 2i) contains the normalizing factors for the right singular vector matrix. See ?lasd8 for details.

z

REAL for slasda

DOUBLE PRECISION for dlasda.

Array,

DIMENSION (*ldu*, *nlvl*) if *icompq* = 1 and

DIMENSION (*n*) if *icompq* = 0. The first *k* elements of *z*(1, *i*) contain the components of the deflation-adjusted updating row vector for subproblems on the *i*-th level.

poles

REAL for slasda

DOUBLE PRECISION for dlasda.

Array, DIMENSION(*ldu*, 2**nlvl*)

if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *poles*(1, 2*i* - 1) and *poles*(1, 2*i*) contain the new and old singular values involved in the secular equations on the *i*-th level.

givptr

INTEGER. Array, DIMENSION (*n*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *givptr*(*i*) records the number of Givens rotations performed on the *i*-th problem on the computation tree.

givcol

INTEGER .

Array, DIMENSION(*ldgcol*, 2**nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givcol*(1, 2 *i* - 1) and *givcol*(1, 2 *i*) record the locations of Givens rotations performed on the *i*-th level on the computation tree.

perm

INTEGER. Array, DIMENSION (*ldgcol*, *nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *perm* (1, *i*) records permutations done on the *i*-th level of the computation tree.

givnum

REAL for slasda

DOUBLE PRECISION for dlasda.

Array DIMENSION (*ldu*, 2**nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givnum*(1, 2 *i* - 1) and *givnum*(1, 2 *i*) record the *C*- and *S*-values of Givens rotations performed on the *i*-th level on the computation tree.

c

REAL for slasda

DOUBLE PRECISION for dlasda.

Array,

DIMENSION (*n*) if *icompq* = 1, and

DIMENSION (1) if *icompq* = 0.

If *icompq* = 1 and the *i*-th subproblem is not square, on exit, *c*(*i*) contains the *C*-value of a Givens rotation related to the right null space of the *i*-th subproblem.

s — REAL for slasda

DOUBLE PRECISION for dlasda.

Array,

DIMENSION (*n*) *icompq* = 1, and

DIMENSION (1) if *icompq* = 0.

If *icompq* = 1 and the *i*-th subproblem is not square, on exit, *s*(*i*) contains the *S*-value of a Givens rotation related to the right null space of the *i*-th subproblem.

info — INTEGER.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value

> 0: If *info* = 1, an singular value did not converge

?lasdq

Computes the SVD of a real bidiagonal matrix with diagonal d and off-diagonal e. Used by ?bdsdc.

Syntax

```
call slasdq( uplo, sqre, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info )
call dlasdq( uplo, sqre, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info )
```

Include Files

- mkl.fi

Description

The routine ?lasdq computes the singular value decomposition (*SVD*) of a real (upper or lower) bidiagonal matrix with diagonal *d* and off-diagonal *e*, accumulating the transformations if desired. If *B* is the input bidiagonal matrix, the algorithm computes orthogonal matrices *Q* and *P* such that $B = Q*S*P^T$. The singular values *S* are overwritten on *d*.

The input matrix *U* is changed to $U*Q$ if desired.

The input matrix *VT* is changed to P^T*VT if desired.

The input matrix *C* is changed to Q^T*C if desired.

Input Parameters

uplo — CHARACTER*1. On entry, *uplo* specifies whether the input bidiagonal matrix is upper or lower bidiagonal.

If *uplo* = 'U' or 'u', *B* is upper bidiagonal;

If *uplo* = 'L' or 'l', *B* is lower bidiagonal.

sqre INTEGER.

= 0: then the input matrix is *n*-by-*n*.

= 1: then the input matrix is *n*-by-(*n*+1) if *uplu* = 'U' and (*n*+1)-by-*n* if *uplu*

= 'L'. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.

n INTEGER. On entry, *n* specifies the number of rows and columns in the matrix. *n* must be at least 0.

ncvt INTEGER. On entry, *ncvt* specifies the number of columns of the matrix *VT*. *ncvt* must be at least 0.

nru INTEGER. On entry, *nru* specifies the number of rows of the matrix *U*. *nru* must be at least 0.

ncc INTEGER. On entry, *ncc* specifies the number of columns of the matrix *C*. *ncc* must be at least 0.

d REAL for slasdq

DOUBLE PRECISION for dlasdq.

Array, DIMENSION (*n*). On entry, *d* contains the diagonal entries of the bidiagonal matrix.

e REAL for slasdq

DOUBLE PRECISION for dlasdq.

Array, DIMENSION is (*n*-1) if *sqre* = 0 and *n* if *sqre* = 1. On entry, the entries of *e* contain the off-diagonal entries of the bidiagonal matrix.

vt REAL for slasdq

DOUBLE PRECISION for dlasdq.

Array, DIMENSION (*ldvt*, *ncvt*). On entry, contains a matrix which on exit has been premultiplied by P^T, dimension *n*-by-*ncvt* if *sqre* = 0 and (*n*+1)-by-*ncvt* if *sqre* = 1 (not referenced if *ncvt*=0).

ldvt INTEGER. On entry, *ldvt* specifies the leading dimension of *vt* as declared in the calling (sub) program. *ldvt* must be at least 1. If *ncvt* is nonzero, *ldvt* must also be at least *n*.

u REAL for slasdq

DOUBLE PRECISION for dlasdq.

Array, DIMENSION (*ldu*, *n*). On entry, contains a matrix which on exit has been postmultiplied by *Q*, dimension *nru*-by-*n* if *sqre* = 0 and *nru*-by-(*n* +1) if *sqre* = 1 (not referenced if *nru*=0).

ldu INTEGER. On entry, *ldu* specifies the leading dimension of *u* as declared in the calling (sub) program. *ldu* must be at least max(1, *nru*) .

c REAL for slasdq

DOUBLE PRECISION for dlasdq.

Array, DIMENSION (*ldc*, *ncc*). On entry, contains an *n*-by-*ncc* matrix which on exit has been premultiplied by *Q*', dimension *n*-by-*ncc* if *sqre* = 0 and (*n*+1)-by-*ncc* if *sqre* = 1 (not referenced if *ncc*=0).

ldc — INTEGER. On entry, *ldc* specifies the leading dimension of *C* as declared in the calling (sub) program. *ldc* must be at least 1. If *ncc* is non-zero, *ldc* must also be at least *n*.

work — REAL for slasdq

DOUBLE PRECISION for dlasdq.

Array, DIMENSION (4*n*). This is a workspace array. Only referenced if one of *ncvt*, *nru*, or *ncc* is nonzero, and if *n* is at least 2.

Output Parameters

d — On normal exit, *d* contains the singular values in ascending order.

e — On normal exit, *e* will contain 0. If the algorithm does not converge, *d* and *e* will contain the diagonal and superdiagonal entries of a bidiagonal matrix orthogonally equivalent to the one given as input.

vt — On exit, the matrix has been premultiplied by *P*'.

u — On exit, the matrix has been postmultiplied by *Q*.

c — On exit, the matrix has been premultiplied by *Q*'.

info — INTEGER. On exit, a value of 0 indicates a successful exit. If *info* < 0, argument number -*info* is illegal. If *info* > 0, the algorithm did not converge, and *info* specifies how many superdiagonals did not converge.

?lasdt

Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.

Syntax

```
call slasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
call dlasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
```

Include Files

- mkl.fi

Description

The routine creates a tree of subproblems for bidiagonal divide and conquer.

Input Parameters

n — INTEGER. On entry, the number of diagonal elements of the bidiagonal matrix.

msub INTEGER. On entry, the maximum row dimension each subproblem at the bottom of the tree can be of.

Output Parameters

lvl INTEGER. On exit, the number of levels on the computation tree.

nd INTEGER. On exit, the number of nodes on the tree.

inode INTEGER.

Array, DIMENSION (*n*). On exit, centers of subproblems.

ndiml INTEGER.

Array, DIMENSION (*n*). On exit, row dimensions of left children.

ndimr INTEGER.

Array, DIMENSION (*n*). On exit, row dimensions of right children.

?laset

Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.

Syntax

```
call slaset( uplo, m, n, alpha, beta, a, lda )
call dlaset( uplo, m, n, alpha, beta, a, lda )
call claset( uplo, m, n, alpha, beta, a, lda )
call zlaset( uplo, m, n, alpha, beta, a, lda )
```

Include Files

- mkl.fi

Description

The routine initializes an *m*-by-*n* matrix *A* to *beta* on the diagonal and *alpha* on the off-diagonals.

Input Parameters

The data types are given for the Fortran interface.

uplo CHARACTER*1. Specifies the part of the matrix *A* to be set.

If *uplo* = 'U', upper triangular part is set; the strictly lower triangular part of *A* is not changed.

If *uplo* = 'L': lower triangular part is set; the strictly upper triangular part of *A* is not changed.

Otherwise: *A*ll of the matrix *A* is set.

m INTEGER. The number of rows of the matrix *A*. $m \geq 0$.

n INTEGER. The number of columns of the matrix *A*.

	$n \geq 0$.
alpha, *beta*	REAL for slaset DOUBLE PRECISION for dlaset COMPLEX for claset DOUBLE COMPLEX for zlaset. The constants to which the off-diagonal and diagonal elements are to be set, respectively.
a	REAL for slaset DOUBLE PRECISION for dlaset COMPLEX for claset DOUBLE COMPLEX for zlaset. Array, DIMENSION (*lda*, *n*). The array *a* contains the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda* $\geq$ max(1,*m*).

Output Parameters

a	On exit, the leading *m*-by-*n* submatrix of *A* is set as follows: if *uplo* = 'U', $A_{ij} = alpha, 1 \leq i \leq j-1, 1 \leq j \leq n$, if *uplo* = 'L', $A_{ij} = alpha, j+1 \leq i \leq m, 1 \leq j \leq n$, otherwise, $A_{ij} = alpha, 1 \leq i \leq m, 1 \leq j \leq n, i \neq j$, and, for all *uplo*, $A_{ii} = beta, 1 \leq i \leq \min(m, n)$.

?lasq1

Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.

Syntax

```
call slasq1( n, d, e, work, info )
call dlasq1( n, d, e, work, info )
```

Include Files

- mkl.fi

Description

The routine ?lasq1 computes the singular values of a real *n*-by-*n* bidiagonal matrix *Z* with diagonal *d* and off-diagonal *e*. The singular values are computed to high relative accuracy, in the absence of denormalization, underflow and overflow.

Input Parameters

n	INTEGER.The number of rows and columns in the matrix. $n \geq 0$.
d	REAL for slasq1 DOUBLE PRECISION for dlasq1. Array, DIMENSION (*n*). On entry, *d* contains the diagonal elements of the bidiagonal matrix whose *SVD* is desired.
e	REAL for slasq1 DOUBLE PRECISION for dlasq1. Array, DIMENSION (*n*). On entry, elements *e*(1:*n*-1) contain the off-diagonal elements of the bidiagonal matrix whose *SVD* is desired.
work	REAL for slasq1 DOUBLE PRECISION for dlasq1. Workspace array, DIMENSION(4*n*).

Output Parameters

d	On normal exit, *d* contains the singular values in decreasing order.
e	On exit, *e* is overwritten.
info	INTEGER. = 0: successful exit; < 0: if *info* = -*i*, the *i*-th argument had an illegal value; > 0: the algorithm failed: = 1, a split was marked by a positive value in *e*; = 2, current block of *Z* not diagonalized after 100*n* iterations (in inner while loop) - on exit the current contents of *d* and *e* represent a matrix with the same singular values as the matrix with which ?lasq1 was originally called, and which the calling subroutine could use to finish the computation, or even feed back into ?lasq1; = 3, termination criterion of outer while loop not met (program created more than *n* unreduced blocks.

?lasq2

Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the quotient difference array z to high relative accuracy. Used by ?bdsqr *and* ?stegr.

Syntax

```
call slasq2( n, z, info )
call dlasq2( n, z, info )
```

Include Files

- mkl.fi

Description

The routine ?lasq2 computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the quotient difference array *z* to high relative accuracy, in the absence of denormalization, underflow and overflow.

To see the relation of *z* to the tridiagonal matrix, let *L* be a unit lower bidiagonal matrix with subdiagonals *z*(2,4,6,,..) and let *U* be an upper bidiagonal matrix with 1's above and diagonal *z*(1,3,5,,..). The tridiagonal is *LU* or, if you prefer, the symmetric tridiagonal to which it is similar.

Input Parameters

n — INTEGER. The number of rows and columns in the matrix. $n \geq 0$.

z — REAL for slasq2

DOUBLE PRECISION for dlasq2.

Array, DIMENSION (4 * *n*).

On entry, *z* holds the quotient difference array.

Output Parameters

z — On exit, entries 1 to *n* hold the eigenvalues in decreasing order, *z*(2*n*+1) holds the trace, and *z*(2*n*+2) holds the sum of the eigenvalues. If $n > 2$, then *z*(2*n*+3) holds the iteration count, *z*(2*n*+4) holds ndivs/nin^2, and *z*(2*n*+5) holds the percentage of shifts that failed.

info — INTEGER.

= 0: successful exit;

< 0: if the *i*-th argument is a scalar and had an illegal value, then *info* = −*i*, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = −(*i**100+ *j*);

> 0: the algorithm failed:

= 1, a split was marked by a positive value in *e*;

= 2, current block of *z* not diagonalized after 100**n* iterations (in inner while loop) - On exit *z* holds a quotient difference array with the same eigenvalues as the *z* array on entry;

= 3, termination criterion of outer while loop not met (program created more than *n* unreduced blocks).

Application Notes

The routine ?lasq2 defines a logical variable, *ieee*, which is .TRUE. on machines which follow ieee-754 floating-point standard in their handling of infinities and NaNs, and .FALSE. otherwise. This variable is passed to ?lasq3.

?lasq3

Checks for deflation, computes a shift and calls dqds.
Used by ?bdsqr.

Syntax

call slasq3(*i0*, *n0*, *z*, *pp*, *dmin*, *sigma*, *desig*, *qmax*, *nfail*, *iter*, *ndiv*, *ieee*, *ttype*, *dmin1*, *dmin2*, *dn*, *dn1*, *dn2*, *g*, *tau*)

call dlasq3(*i0*, *n0*, *z*, *pp*, *dmin*, *sigma*, *desig*, *qmax*, *nfail*, *iter*, *ndiv*, *ieee*, *ttype*, *dmin1*, *dmin2*, *dn*, *dn1*, *dn2*, *g*, *tau*)

Include Files

- mkl.fi

Description

The routine ?lasq3 checks for deflation, computes a shift *tau*, and calls *dqds*. In case of failure, it changes shifts, and tries again until output is positive.

Input Parameters

i0	INTEGER. First index.
n0	INTEGER. Last index.
z	REAL for slasq3 DOUBLE PRECISION for dlasq3. Array, DIMENSION (4*n*). *z* holds the *qd* array.
pp	INTEGER. *pp*=0 for ping, *pp*=1 for pong. *pp*=2 indicates that flipping was applied to the *Z* array and that the initial tests for deflation should not be performed.
desig	REAL for slasq3 DOUBLE PRECISION for dlasq3. Lower order part of *sigma*.
qmax	REAL for slasq3 DOUBLE PRECISION for dlasq3. Maximum value of *q*.
ieee	LOGICAL. Flag for ieee or non-ieee arithmetic (passed to ?lasq5).
ttype	INTEGER. Shift type.
dmin1, *dmin2*, *dn*, *dn1*, *dn2*, *g*, *tau*	REAL for slasq3 DOUBLE PRECISION for dlasq3.

	These scalars are passed as arguments in order to save their values between calls to `?lasq3`.

Output Parameters

dmin	`REAL` for `slasq3` `DOUBLE PRECISION` for `dlasq3`. Minimum value of *d*.
pp	`INTEGER`. *pp*=0 for ping, *pp*=1 for pong. *pp*=2 indicates that flipping was applied to the *Z* array and that the initial tests for deflation should not be performed.
sigma	`REAL` for `slasq3` `DOUBLE PRECISION` for `dlasq3`. Sum of shifts used in the current segment.
desig	Lower order part of *sigma*.
nfail	`INTEGER`. Number of times shift was too big.
iter	`INTEGER`. Number of iterations.
ndiv	`INTEGER`. Number of divisions.
ttype	`INTEGER`. Shift type.
dmin1, *dmin2*, *dn*, *dn1*, *dn2*, *g*, *tau*	`REAL` for `slasq3` `DOUBLE PRECISION` for `dlasq3`. These scalars are passed as arguments in order to save their values between calls to `?lasq3`.

?lasq4

Computes an approximation to the smallest eigenvalue using values of d from the previous transform. Used by `?bdsqr`.

Syntax

```
call slasq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau, ttype, g )
call dlasq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau, ttype, g )
```

Include Files

- `mkl.fi`

Description

The routine computes an approximation *tau* to the smallest eigenvalue using values of d from the previous transform.

Input Parameters

i0 — INTEGER. First index.

n0 — INTEGER. Last index.

z — REAL for slasq4

DOUBLE PRECISION for dlasq4.

Array, DIMENSION (4*n*).

z holds the *qd* array.

pp — INTEGER. *pp*=0 for ping, *pp*=1 for pong.

n0in — INTEGER. The value of *n0* at start of eigtest.

dmin — REAL for slasq4

DOUBLE PRECISION for dlasq4.

Minimum value of *d*.

dmin1 — REAL for slasq4

DOUBLE PRECISION for dlasq4.

Minimum value of *d*, excluding *d*(*n0*).

dmin2 — REAL for slasq4

DOUBLE PRECISION for dlasq4.

Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).

dn — REAL for slasq4

DOUBLE PRECISION for dlasq4. Contains *d*(*n*).

dn1 — REAL for slasq4

DOUBLE PRECISION for dlasq4. Contains *d*(*n*-1).

dn2 — REAL for slasq4

DOUBLE PRECISION for dlasq4. Contains *d*(*n*-2).

g — REAL for slasq4

DOUBLE PRECISION for dlasq4.

A scalar passed as an argument in order to save its value between calls to ?lasq4.

Output Parameters

tau — REAL for slasq4

DOUBLE PRECISION for dlasq4.

Shift.

ttype — INTEGER. Shift type.

g — REAL for slasq4

DOUBLE PRECISION for dlasq4.

A scalar passed as an argument in order to save its value between calls to ?lasq4.

?lasq5

Computes one dqds transform in ping-pong form.
Used by ?bdsqr *and* ?stegr.

Syntax

```
call slasq5( i0, n0, z, pp, tau, sigma, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee, eps )
call dlasq5( i0, n0, z, pp, tau, sigma, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee, eps )
```

Include Files

- mkl.fi

Description

The routine computes one dqds transform in ping-pong form: one version for ieee machines, another for non-ieee machines.

Input Parameters

i0 — INTEGER. First index.

n0 — INTEGER. Last index.

z — REAL for slasq5

DOUBLE PRECISION for dlasq5.

Array, DIMENSION (4*n*). *z* holds the qd array. *emin* is stored in *z*(4**n0*) to avoid an extra argument.

pp — INTEGER. *pp*=0 for ping, *pp*=1 for pong.

tau — REAL for slasq5

DOUBLE PRECISION for dlasq5.

This is the shift.

sigma — REAL for slasq5

DOUBLE PRECISION for dlasq5.

This is the accumulated shift up to the current point.

ieee — LOGICAL. Flag for IEEE or non-IEEE arithmetic.

eps — REAL for slasq5

DOUBLE PRECISION for dlasq5.

This is the value of epsilon used.

Output Parameters

dmin
REAL for slasq5
DOUBLE PRECISION for dlasq5.
Minimum value of *d*.

dmin1
REAL for slasq5
DOUBLE PRECISION for dlasq5.
Minimum value of *d*, excluding *d*(*n0*).

dmin2
REAL for slasq5
DOUBLE PRECISION for dlasq5.
Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).

dn
REAL for slasq5
DOUBLE PRECISION for dlasq5. Contains *d*(*n0*), the last value of *d*.

dnm1
REAL for slasq5
DOUBLE PRECISION for dlasq5. Contains *d*(*n0*-1).

dnm2
REAL for slasq5
DOUBLE PRECISION for dlasq5. Contains *d*(*n0*-2).

?lasq6

Computes one dqd transform in ping-pong form. Used by ?bdsqr *and* ?stegr.

Syntax

```
call slasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
call dlasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
```

Include Files

- mkl.fi

Description

The routine ?lasq6 computes one *dqd* (shift equal to zero) transform in ping-pong form, with protection against underflow and overflow.

Input Parameters

i0
INTEGER. First index.

n0
INTEGER. Last index.

z
REAL for slasq6
DOUBLE PRECISION for dlasq6.

Array, DIMENSION (4*n*). *Z* holds the qd array. *emin* is stored in `z(4*n0)` to avoid an extra argument.

pp — INTEGER. *pp*=0 for ping, *pp*=1 for pong.

Output Parameters

dmin — REAL for slasq6

DOUBLE PRECISION for dlasq6.

Minimum value of *d*.

dmin1 — REAL for slasq6

DOUBLE PRECISION for dlasq6.

Minimum value of *d*, excluding *d*(*n0*).

dmin2 — REAL for slasq6

DOUBLE PRECISION for dlasq6.

Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).

dn — REAL for slasq6

DOUBLE PRECISION for dlasq6. Contains *d*(*n0*), the last value of *d*.

dnm1 — REAL for slasq6

DOUBLE PRECISION for dlasq6. Contains *d*(*n0*-1).

dnm2 — REAL for slasq6

DOUBLE PRECISION for dlasq6. Contains *d*(*n0*-2).

?lasr

Applies a sequence of plane rotations to a general rectangular matrix.

Syntax

```
call slasr( side, pivot, direct, m, n, c, s, a, lda )
call dlasr( side, pivot, direct, m, n, c, s, a, lda )
call clasr( side, pivot, direct, m, n, c, s, a, lda )
call zlasr( side, pivot, direct, m, n, c, s, a, lda )
```

Include Files

- mkl.fi

Description

The routine applies a sequence of plane rotations to a real/complex matrix *A*, from the left or the right.

A := *P***A*, when *side* = 'L' (Left-hand side)

A := *A***P*', when *side* = 'R' (Right-hand side)

where *P* is an orthogonal matrix consisting of a sequence of plane rotations with *z* = *m* when `side = 'L'` and *z* = *n* when `side = 'R'`.

When `direct = 'F'` (Forward sequence), then

$P = P(z-1)*...P(2)*P(1)$,

and when `direct = 'B'` (Backward sequence), then

$P = P(1)*P(2)*...*P(z-1)$,

where *P*(*k*) is a plane rotation matrix defined by the 2-by-2 plane rotation:

$$R(k) = \begin{bmatrix} c(k) & s(k) \\ -s(k) & c(k) \end{bmatrix}$$

When `pivot = 'V'` (Variable pivot), the rotation is performed for the plane (*k*, *k* + 1), that is, *P*(*k*) has the form

$$P(k) = \begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & 1 & & & & \\ & & & c(k) & s(k) & & \\ & & & -s(k) & c(k) & & \\ & & & & & 1 & \\ & & & & & & \cdots \\ & & & & & & & 1 \end{bmatrix}$$

where *R*(*k*) appears as a rank-2 modification to the identity matrix in rows and columns *k* and *k*+1.

When `pivot = 'T'` (Top pivot), the rotation is performed for the plane `(1,k+1)`, so *P*(*k*) has the form

$$P(k) = \begin{bmatrix} c(k) & & & s(k) & & \\ & 1 & & & & \\ & & \cdots & & & \\ & & & 1 & & \\ -s(k) & & & c(k) & & \\ & & & & 1 & \\ & & & & & \cdots \\ & & & & & & 1 \end{bmatrix}$$

where $R(k)$ appears in rows and columns k and $k+1$.

Similarly, when *pivot* = 'B' (Bottom pivot), the rotation is performed for the plane (k, z), giving $P(k)$ the form

$$P(k) = \begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & 1 & & & & \\ & & & c(k) & & & s(k) \\ & & & & 1 & & \\ & & & & & \cdots & \\ & & & & & & 1 \\ & & & -s(k) & & & c(k) \end{bmatrix}$$

where $R(k)$ appears in rows and columns k and z. The rotations are performed without ever forming $P(k)$ explicitly.

Input Parameters

side — CHARACTER*1. Specifies whether the plane rotation matrix P is applied to A on the left or the right.

= 'L': left, compute $A := P*A$

= 'R': right, compute $A := A*P'$

direct CHARACTER*1. Specifies whether *P* is a forward or backward sequence of plane rotations.

= 'F': forward, $P = P(z-1)*...*P(2)*P(1)$

= 'B': backward, $P = P(1)*P(2)*...*P(z-1)$

pivot CHARACTER*1. Specifies the plane for which *P*(*k*) is a plane rotation matrix.

= 'V': Variable pivot, the plane (*k*, *k*+1)

= 'T': Top pivot, the plane (1, *k*+1)

= 'B': Bottom pivot, the plane (*k*, *z*)

m INTEGER. The number of rows of the matrix *A*.

If $m \leq 1$, an immediate return is effected.

n INTEGER. The number of columns of the matrix *A*.

If $n \leq 1$, an immediate return is effected.

c, *s* REAL for slasr/clasr

DOUBLE PRECISION for dlasr/zlasr.

Arrays, DIMENSION

(*m*-1) if *side* = 'L',

(*n*-1) if *side* = 'R' .

c(*k*) and *s*(*k*) contain the cosine and sine of the plane rotations respectively that define the 2-by-2 plane rotation part (*R*(*k*)) of the *P*(*k*) matrix as described above in *Description*.

a REAL for slasr

DOUBLE PRECISION for dlasr

COMPLEX for clasr

DOUBLE COMPLEX for zlasr.

Array, DIMENSION (*lda*, *n*).

The *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.

$lda \geq \max(1,m)$.

Output Parameters

a On exit, *A* is overwritten by *P***A* if *side* = 'R', or by *A***P*' if *side* = 'L'.

?lasrt

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt( id, n, d, info )
call dlasrt( id, n, d, info )
```

Include Files

- mkl.fi

Description

The routine ?lasrt sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20. Dimension of stack limits *n* to about 2^{32}.

Input Parameters

The data types are given for the Fortran interface.

id	CHARACTER*1. (*d*(1) ≤ ... ≤ *d*(*n*)) or into decreasing order (*d*(1) ≥ ... ≥*d*(*n*)), depending on *id*.
n	INTEGER. The length of the array *d*.
d	REAL for slasrt DOUBLE PRECISION for dlasrt. On entry, the array to be sorted.

Output Parameters

d	On exit, *d* has been sorted into increasing order (*d*[0]≤d[1]≤ ... ≤ *d*[n-1]) or into decreasing order (*d*[0] ≥ d[1] ≥ ... ≥*d*[*n*-1]), depending on *id*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* < 0, the *i*-th parameter had an illegal value.

?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call slassq( n, x, incx, scale, sumsq )
call dlassq( n, x, incx, scale, sumsq )
call classq( n, x, incx, scale, sumsq )
call zlassq( n, x, incx, scale, sumsq )
```

Include Files

- mkl.fi

Description

The real routines slassq/dlassq return the values *scl* and *smsq* such that

$scl^2 * smsq = x(1)^2 + ... + x(n)^2 + scale^2 * sumsq$,

where $x(i) = x(1 + (i - 1) incx)$.

The value of *sumsq* is assumed to be non-negative and *scl* returns the value

$scl = \max(scale, \text{abs}(x(i)))$.

Values *scale* and *sumsq* must be supplied in *scale* and *sumsq*, and *scl* and *smsq* are overwritten on *scale* and *sumsq*, respectively.

The complex routines classq/zlassq return the values *scl* and *ssq* such that

$scl^2 * ssq = x(1)^2 + ... + x(n)^2 + scale^2 * sumsq$,

where $x(i) = \text{abs}(x(1 + (i - 1) * incx))$.

The value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy $1.0 \le ssq \le sumsq + 2n$

scale is assumed to be non-negative and *scl* returns the value

$scl = \max(scale, \text{abs}(\text{real}(x(i))), \text{abs}(\text{aimag}(x(i))))$.

Values *scale* and *sumsq* must be supplied in *scale* and *sumsq*, and *scl* and *ssq* are overwritten on *scale* and *sumsq*, respectively.

All routines ?lassq make only one pass through the vector *x*.

Input Parameters

n	INTEGER. The number of elements to be used from the vector *x*.
x	REAL for slassq DOUBLE PRECISION for dlassq COMPLEX for classq DOUBLE COMPLEX for zlassq. The vector for which a scaled sum of squares is computed: $x(i) = x(1+(i-1)*incx), 1 \le i \le n$.
incx	INTEGER. The increment between successive values of the vector *x*. $incx > 0$.
scale	REAL for slassq/classq DOUBLE PRECISION for dlassq/zlassq. On entry, the value *scale* in the equation above.
sumsq	REAL for slassq/classq DOUBLE PRECISION for dlassq/zlassq. On entry, the value *sumsq* in the equation above.

Output Parameters

scale	On exit, *scale* is overwritten with *scl*, the scaling factor for the sum of squares.
sumsq	For real flavors: On exit, *sumsq* is overwritten with the value *smsq* in the equation above. For complex flavors: On exit, *sumsq* is overwritten with the value *ssq* in the equation above.

?lasv2

Computes the singular value decomposition of a 2-by-2 triangular matrix.

Syntax

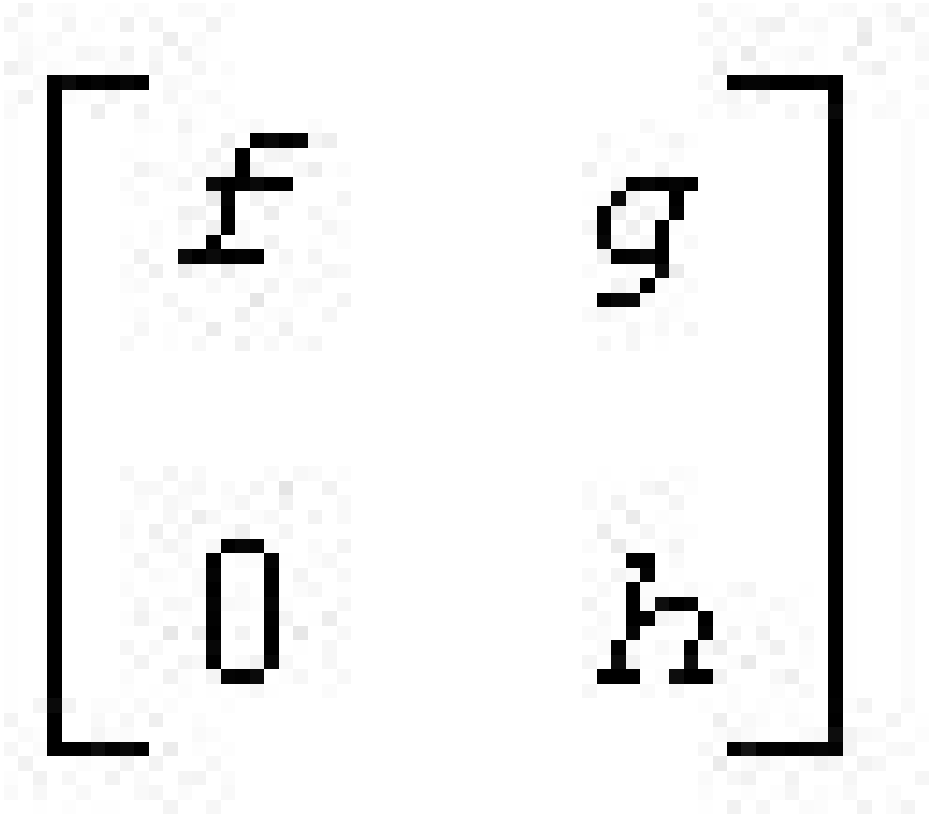
```
call slasv2( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
call dlasv2( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
```

Include Files

- mkl.fi

Description

The routine ?lasv2 computes the singular value decomposition of a 2-by-2 triangular matrix



On return, abs(*ssmax*) is the larger singular value, abs(*ssmin*) is the smaller singular value, and (*csl*,*snl*) and (*csr*,*snr*) are the left and right singular vectors for abs(*ssmax)*, giving the decomposition

$$\begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} f & g \\ 0 & h \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix} = \begin{bmatrix} ssmax & 0 \\ 0 & ssmin \end{bmatrix}$$

Input Parameters

f, *g*, *h*

REAL for slasv2

DOUBLE PRECISION for dlasv2.

The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

ssmin, *ssmax*

REAL for slasv2

DOUBLE PRECISION for dlasv2.

abs(*ssmin*) and abs(*ssmax*) is the smaller and the larger singular value, respectively.

snl, *csl*

REAL for slasv2

DOUBLE PRECISION for dlasv2.

The vector (*csl*, *snl*) is a unit left singular vector for the singular value abs(*ssmax*).

snr, *csr*

REAL for slasv2

DOUBLE PRECISION for dlasv2.

The vector (*csr*, *snr*) is a unit right singular vector for the singular value abs(*ssmax*).

Application Notes

Any input parameter may be aliased with any output parameter.

Barring over/underflow and assuming a guard digit in subtraction, all output quantities are correct to within a few units in the last place (ulps).

In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?laswlq

Computes blocked Short-Wide LQ matrix factorization.

```
call slaswlq(m, n, mb, nb, a, lda, t, ldt, work, lwork, info)
call dlaswlq(m, n, mb, nb, a, lda, t, ldt, work, lwork, info)
call claswlq(m, n, mb, nb, a, lda, t, ldt, work, lwork, info)
call zlaswlq(m, n, mb, nb, a, lda, t, ldt, work, lwork, info)
```

Description

?laswlq computes a blocked Short-Wide LQ (SWLQ) factorization of an *m*-by-*n* matrix *A*, where *n*≥*m*: *A* = *L***Q*.

SWLQ performs LQ by a sequence of orthogonal transformations, representing *Q* as a product of other orthogonal matrices

Q = *Q*(1) * *Q*(2) * . . . * *Q*(*k*)

where each *Q*(*i*) zeros out upper diagonal entries of a block of *nb* rows of *A*:

Q(1) zeros out the upper diagonal entries of rows 1:*nb* of *A*,

Q(2) zeros out the bottom *mb* - *n* rows of rows [1:*m*, *nb* + 1:2**nb* - *m*] of *A*,

Q(3) zeros out the bottom *mb* - *n* rows of rows [1:*m*, 2**nb* - *m* + 1:3**nb* - 2**m*] of *A*....

Q(1) is computed bygelqt, which represents *Q*(1) by Householder vectors stored under the diagonal of rows 1:*mb* of *a*, and by upper triangular block reflectors, stored in array *t*(1:*ldt*, 1:*n*). For more information, see gelqt.

Q(*i*) for *i* > 1 is computed by tplqt, which represents *Q*(*i*) by Householder vectors stored in columns [(*i* - 1)*(*nb* - *m*) + *m* + 1:*i**(*nb* - *m*) + *m*] of *a*, and by upper triangular block reflectors, stored in array *t*(1:*ldt*, (*i* - 1)**m* + 1:*i***m*). The last *Q*(*k*) may use fewer rows. For more information, see tplqt. For more details of the overall algorithm, see [DEMMEL12].

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. *m*≥ 0.
n	INTEGER. The number of columns of the matrix *A*. *n*≥*m*≥ 0.
mb	INTEGER. The row block size to be used in the blocked QR. *m*≥*mb*≥ 1
nb	INTEGER. The column block size to be used in the blocked QR. *nb* > *m*.
a	REAL for slaswlq DOUBLE PRECISION for dlaswlq COMPLEX for claswlq COMPLEX*16 for zlaswlq Array of size (*lda*, *n*). On entry, the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*m*).
ldt	INTEGER. The leading dimension of the array *t*. *ldt*≥*mb*.
lwork	INTEGER. The dimension of the array *work*. *lwork*≥*mb* * *m*. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by XERBLA.

Output Parameters

a	On exit, the elements on and below the diagonal of the array contain the *n*-by-*n* lower triangular matrix *L*; the elements above the diagonal represent *Q* by the rows of blocked *V*.
t	REAL for slaswlq DOUBLE PRECISION for dlaswlq COMPLEX for claswlq COMPLEX*16 for zlaswlq Array of size (*ldt*, *n* * *Number_of_row_blocks*), where *Number_of_row_blocks* = ceiling((*n* - *m*)/(*nb*-*m*)). The blocked upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks.
work	REAL for slaswlq DOUBLE PRECISION for dlaswlq COMPLEX for claswlq COMPLEX*16 for zlaswlq Workspace array of size (max(1, *lwork*)).
info	INTEGER. *info* = 0: successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call slaswp( n, a, lda, k1, k2, ipiv, incx )
call dlaswp( n, a, lda, k1, k2, ipiv, incx )
call claswp( n, a, lda, k1, k2, ipiv, incx )
call zlaswp( n, a, lda, k1, k2, ipiv, incx )
```

Include Files

- mkl.fi

Description

The routine performs a series of row interchanges on the matrix *A*. One row interchange is initiated for each of rows *k1* through *k2* of *A*.

Input Parameters

The data types are given for the Fortran interface.

n	INTEGER. The number of columns of the matrix *A*.
a	REAL for slaswp DOUBLE PRECISION for dlaswp COMPLEX for claswp DOUBLE COMPLEX for zlaswp. Array, size *lda* by *n*. Array *a* contains the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*.
k1	INTEGER. The first element of *ipiv* for which a row interchange will be done.
k2	INTEGER. The last element of *ipiv* for which a row interchange will be done.
ipiv	INTEGER. Array, size *k1*+(*k2*–*k1*)*|*incx*|). The vector of pivot indices. Only the elements in positions *k1* through *k2* of *ipiv* are accessed. *ipiv*(*k*) = *l* implies rows *k* and *l* are to be interchanged.
incx	INTEGER. The increment between successive values of *ipiv*. If *ipiv* is negative, the pivots are applied in reverse order.

Output Parameters

a	On exit, the permuted matrix.

?lasy2

Solves the Sylvester matrix equation where the matrices are of order 1 or 2.

Syntax

```
call slasy2( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr, ldtr, b, ldb, scale, x, ldx, xnorm, info )
call dlasy2( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr, ldtr, b, ldb, scale, x, ldx, xnorm, info )
```

Include Files

- mkl.fi

Description

The routine solves for the *n1*-by-*n2* matrix *X*, 1 ≤ *n1*, *n2* ≤ 2, in

op(*TL*)*X + *isgn**X*op(*TR*) = *scale**B*,

where

TL is *n1*-by-*n1*,

TR is *n2*-by-*n2*,

B is *n1*-by-*n2*,

and *isgn* = 1 or -1. op(*T*) = *T* or T^T, where T^T denotes the transpose of *T*.

Input Parameters

ltranl
LOGICAL.
On entry, *ltranl* specifies the op(*TL*):
= .FALSE., op(*TL*) = *TL*,
= .TRUE., op(*TL*) = $(TL)^T$.

ltranr
LOGICAL.
On entry, *ltranr* specifies the op(*TR*):
= .FALSE., op(*TR*) = *TR*,
= .TRUE., op(*TR*) = $(TR)^T$.

isgn
INTEGER. On entry, *isgn* specifies the sign of the equation as described before. *isgn* may only be 1 or -1.

n1
INTEGER. On entry, *n1* specifies the order of matrix *TL*.
n1 may only be 0, 1 or 2.

n2
INTEGER. On entry, *n2* specifies the order of matrix *TR*.
n2 may only be 0, 1 or 2.

tl
REAL for slasy2
DOUBLE PRECISION for dlasy2.
Array, DIMENSION (*ldtl*,2).
On entry, *tl* contains an *n1*-by-*n1* matrix *TL*.

ldtl
INTEGER.The leading dimension of the matrix *TL*.
ldtl≥ max(1,*n1*).

tr
REAL for slasy2
DOUBLE PRECISION for dlasy2.
Array, DIMENSION (*ldtr*,2). On entry, *tr* contains an *n2*-by-*n2* matrix *TR*.

ldtr
INTEGER. The leading dimension of the matrix *TR*.
ldtr≥ max(1,*n2*).

b
REAL for slasy2
DOUBLE PRECISION for dlasy2.
Array, DIMENSION (*ldb*,2). On entry, the *n1*-by-*n2* matrix *B* contains the right-hand side of the equation.

ldb
INTEGER. The leading dimension of the matrix *B*.
ldb≥ max(1,*n1*).

ldx	INTEGER. The leading dimension of the output matrix *X*. *ldx*≥ max(1,*n1*).

Output Parameters

scale	REAL for slasy2 DOUBLE PRECISION for dlasy2. On exit, *scale* contains the scale factor. *scale* is chosen less than or equal to 1 to prevent the solution overflowing.
x	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (*ldx*,2). On exit, *x* contains the *n1*-by-*n2* solution.
xnorm	REAL for slasy2 DOUBLE PRECISION for dlasy2. On exit, *xnorm* is the infinity-norm of the solution.
info	INTEGER. On exit, *info* is set to 0: successful exit. 1: *TL* and *TR* have too close eigenvalues, so *TL* or *TR* is perturbed to get a nonsingular equation.

NOTE
For higher speed, this routine does not check the inputs for errors.

?lasyf

Computes a partial factorization of a symmetric matrix, using the diagonal pivoting method.

Syntax

```
call slasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call dlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call clasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- mkl.fi

Description

The routine ?*lasyf* computes a partial factorization of a symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

If *uplo* = 'U':

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^{T} & U_{22}^{T} \end{bmatrix}$$

uplo = 'L'

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^{T} & L_{21}^{T} \\ 0 & I \end{bmatrix}$$

where the order of *D* is at most *nb*.

The actual order is returned in the argument *kb*, and is either *nb* or *nb*-1, or *n* if $n \leq nb$.

This is an auxiliary routine called by `?sytrf`. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if *uplo* = 'U') or A_{22} (if *uplo* = 'L').

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n

INTEGER. The order of the matrix *A*. $n \geq 0$.

nb

INTEGER. The maximum number of columns of the matrix *A* that should be factored. *nb* should be at least 2 to allow for 2-by-2 pivot blocks.

a

REAL for `slasyf`

DOUBLE PRECISION for `dlasyf`

COMPLEX for `clasyf`

DOUBLE COMPLEX for `zlasyf`.

Array, DIMENSION (*lda*, *n*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda

INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

w

REAL for `slasyf`

DOUBLE PRECISION for `dlasyf`

COMPLEX for clasyf

DOUBLE COMPLEX for zlasyf.

Workspace array, DIMENSION (*ldw*, *nb*).

ldw INTEGER. The leading dimension of the array *w*. *ldw* ≥ max(1,*n*).

Output Parameters

kb INTEGER. The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if *n* ≤ *nb*.

a On exit, *a* contains details of the partial factorization.

ipiv INTEGER. Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*.

If *uplo* = 'U', only the last *kb* elements of *ipiv* are set;

if *uplo* = 'L', only the first *kb* elements are set.

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and *D*(*k*, *k*) is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*–1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) were interchanged and *D*(*k*–1:*k*, *k*–1:*k*) is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, then rows and columns *k* +1 and -*ipiv*(*k*) were interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

info INTEGER.

= 0: successful exit

> 0: if *info* = *k*, *D*(*k*, *k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?lasyf_aa

Factorizes a panel of a real or complex symmetric matrix using Aasen's algorithm.

```
call slasyf_aa(uplo, j1, m, nb, a, lda, ipiv, h, ldh, work)
call dlasyf_aa(uplo, j1, m, nb, a, lda, ipiv, h, ldh, work)
call clasyf_aa(uplo, j1, m, nb, a, lda, ipiv, h, ldh, work)
call zlasyf_aa(uplo, j1, m, nb, a, lda, ipiv, h, ldh, work)
```

Description

?lasyf_aa factorizes a panel of a real or complex symmetric matrix *A* using Aasen's algorithm. The panel consists of a set of *nb* rows of *A* when *uplo* is 'U', or a set of *nb* columns when *uplo* is 'L'. In order to factorize the panel, Aasen's algorithm requires the last row, or column, of the previous panel. The first row, or column, of *A* is set to be the first row, or column, of an identity matrix, which is used to factorize the first panel. The resulting *j*-th row of *U*, or *j*-th column of *L*, is stored in the (*j*-1)-st row, or column, of *A* (without the unit diagonals), while the diagonal and subdiagonal of *A* are overwritten by those of *T*.

Input Parameters

uplo	CHARACTER*1. If *uplo* = 'U': Upper triangular of *A* is stored. If *uplo*= 'L': Lower triangular of *A* is stored..
j1	INTEGER. The location of the first row, or column, of the panel within the submatrix of *A*, passed to this routine: when called by ?sytrf_aa, for the first panel, *j1* is 1, while for the remaining panels, *j1* is 2.
m	INTEGER. The size of the submatrix. $m \geq 0$.
nb	INTEGER. The size of the panel to be factorized.
a	REAL for slasyf_aa DOUBLE PRECISION for dlasyf_aa REAL for clasyf_aa COMPLEX*16 for zlasyf_aa Array of size (*lda*, *m*) for the first panel, and size (*lda*,*m* + 1) for the remaining panels. On entry, *a* contains the last row, or column, of the previous panel, and the trailing submatrix of *a* to be factorized, except for the first panel, when only the panel is passed.
lda	INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.
h	REAL for slasyf_aa DOUBLE PRECISION for dlasyf_aa REAL for clasyf_aa COMPLEX*16 for zlasyf_aa Workspace array of size (*ldh*,*nb*).
ldh	INTEGER. The leading dimension of the workspace *h*. $ldh \geq \max(1, m)$.

Output Parameters

a	On exit, the leading panel is factorized.
ipiv	INTEGER . Array of size (*n*). Details of the row and column interchanges: the row and column *k* were interchanged with the row and column *ipiv*(*k*).
work	REAL for slasyf_aa DOUBLE PRECISION for dlasyf_aa REAL for clasyf_aa COMPLEX*16 for zlasyf_aa Workspace array of size(*m*).

?lasyf_rook

Computes a partial factorization of a complex symmetric matrix, using the bounded Bunch-Kaufman diagonal pivoting method.

Syntax

```
call slasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call dlasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call clasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- mkl.fi

Description

The routine ?*lasyf_rook* computes a partial factorization of a complex symmetric matrix *A* using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The partial factorization has the form:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^{T} & U_{22}^{T} \end{bmatrix}$$

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^{T} & L_{21}^{T} \\ 0 & I \end{bmatrix}$$

where the order of *D* is at most *nb*.

The actual order is returned in the argument *kb*, and is either *nb* or *nb*-1, or *n* if $n \leq nb$.

This is an auxiliary routine called by ?sytrf_rook. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if *uplo* = 'U') or A_{22} (if *uplo* = 'L').

Input Parameters

uplo — CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n INTEGER. The order of the matrix *A*. *n* ≥ 0.

nb INTEGER. The maximum number of columns of the matrix *A* that should be factored. *nb* should be at least 2 to allow for 2-by-2 pivot blocks.

a REAL for slasyf_rook

DOUBLE PRECISION for dlasyf_rook

COMPLEX for clasyf_rook

DOUBLE COMPLEX for zlasyf_rook.

Array, DIMENSION (*lda*, *n*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

w REAL for slasyf_rook

DOUBLE PRECISION for dlasyf_rook

COMPLEX for clasyf_rook

DOUBLE COMPLEX for zlasyf_rook.

Workspace array, DIMENSION (*ldw*, *nb*).

ldw INTEGER. The leading dimension of the array *w*. *ldw* ≥ max(1,*n*).

Output Parameters

kb INTEGER. The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if *n* ≤ *nb*.

a On exit, *a* contains details of the partial factorization.

ipiv INTEGER. Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*.

If *uplo* = 'U', only the last *kb* elements of *ipiv* are set;

if *uplo* = 'L', only the first *kb* elements are set.

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) < 0 and *ipiv*(*k* – 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* - 1 and -*ipiv*(*k* - 1) were interchanged, and $D_{k-1:k,\ k-1:k}$ is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) < 0 and *ipiv*(*k* + 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* + 1 and -*ipiv*(*k* + 1) were interchanged, and $D_{k:k+1,\ k:k+1}$ is a 2-by-2 diagonal block.

info INTEGER.

= 0: successful exit

> 0: if *info* = *k*, *D*(*k*, *k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?lahef

Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.

Syntax

```
call clahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- mkl.fi

Description

The routine ?lahef computes a partial factorization of a complex Hermitian matrix *A*, using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

If *uplo* = 'U':

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^{H} & U_{22}^{H} \end{bmatrix}$$

If *uplo* = 'U':

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^{H} & L_{21}^{H} \\ 0 & I \end{bmatrix}$$

where the order of *D* is at most *nb*.

The actual order is returned in the argument *kb*, and is either *nb* or *nb*-1, or *n* if $n \leq nb$.

Note that U^H denotes the conjugate transpose of *U*.

This is an auxiliary routine called by ?hetrf. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if *uplo* = 'U') or A_{22} (if *uplo* = 'L').

Input Parameters

uplo — CHARACTER*1.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:

= 'U': upper triangular

= 'L': lower triangular

n INTEGER. The order of the matrix *A*. *n* ≥ 0.

nb INTEGER. The maximum number of columns of the matrix *A* that should be factored. *nb* should be at least 2 to allow for 2-by-2 pivot blocks.

a COMPLEX for clahef

DOUBLE COMPLEX for zlahef.

Array, DIMENSION (*lda*, *n*).

On entry, the Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

w COMPLEX for clahef

DOUBLE COMPLEX for zlahef.

Workspace array, DIMENSION (*ldw*, *nb*).

ldw INTEGER. The leading dimension of the array *w*. *ldw* ≥ max(1,*n*).

Output Parameters

kb INTEGER. The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if *n* ≤ *nb*.

a On exit, *A* contains details of the partial factorization.

ipiv INTEGER.

Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*.

If *uplo* = 'U', only the last *kb* elements of *ipiv* are set;

if *uplo* = 'L', only the first *kb* elements are set.

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) are interchanged and *D*(*k*, *k*) is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*–1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) are interchanged and *D*(*k*–1:*k*, *k*–1:*k*) is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, then rows and columns *k* +1 and -*ipiv*(*k*) are interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

info INTEGER.

= 0: successful exit

> 0: if *info* = *k*, *D*(*k*, *k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?lahef_aa

Factorizes a panel of a complex Hermitian matrix A using Aasen's algorithm.

```
call clahef_aa(uplo, j1, m, nb, a, lda, ipiv, h, ldh, work)
call zlahef_aa(uplo, j1, m, nb, a, lda, ipiv, h, ldh, work)
```

Description

?lahef_aa factorizes a panel of a complex Hermitian matrix *A* using Aasen's algorithm. The panel consists of a set of *nb* rows of *A* when *uplo* is 'U', or a set of *nb* columns when *uplo* is 'L'. In order to factorize the panel, Aasen's algorithm requires the last row, or column, of the previous panel. The first row, or column, of *A* is set to be the first row, or column, of an identity matrix, which is used to factorize the first panel. The resulting *j*-th row of *U*, or *j*-th column of *L*, is stored in the (*j*-1)-st row, or column, of *A* (without the unit diagonals), while the diagonal and subdiagonal of *A* are overwritten by those of *T*.

Input Parameters

uplo
CHARACTER*1.
If *uplo* = 'U': Upper triangular of *A* is stored.
If *uplo*= 'L': Lower triangular of *A* is stored..

j1
INTEGER. The location of the first row, or column, of the panel within the submatrix of *A*, passed to this routine, for example when called by ?hetrf_aa. For the first panel, *j1* is 1, while for the remaining panels, *j1* is 2.

m
INTEGER. The dimension of the submatrix. *m*≥ 0.

nb
INTEGER. The dimension of the panel to be factorized.

a
COMPLEX for clahef_aa
COMPLEX*16 for zlahef_aa
Array of size (*lda*,*m*) for the first panel, and size (*lda*,*m*+1) for the remaining panels. On entry, *a* contains the last row, or column, of the previous panel, and the trailing submatrix of *A* to be factorized, except for the first panel, only the panel is passed.

lda
INTEGER. The leading dimension of the array *a*. *lda*≥ max(1, *n*).

h
COMPLEX for clahef_aa
COMPLEX*16 for zlahef_aa
Workspace array of size (*ldh*, *nb*).

ldh
INTEGER. The leading dimension of the workspace *h*. *ldh*≥ max(1,*m*).

Output Parameters

a	On exit, the leading panel is factorized.
ipiv	INTEGER . Array of size (*n*). Details of the row and column interchanges: the row and column *k* were interchanged with the row and column *ipiv*(*k*).
h	Workspace array.
work	COMPLEX for clahef_aa COMPLEX*16 for zlahef_aa Workspace array of size (*m*).

?lahef_rook

Computes a partial factorization of a complex Hermitian indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method.

Syntax

```
call clahef_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlahef_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- mkl.fi

Description

The routine ?lahef_rook computes a partial factorization of a complex Hermitian matrix *A*, using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The partial factorization has the form:

If *uplo* = 'U':

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^{H} & U_{22}^{H} \end{bmatrix}$$

If *uplo* = 'L':

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^{H} & L_{21}^{H} \\ 0 & I \end{bmatrix}$$

where the order of *D* is at most *nb*.

The actual order is returned in the argument *kb*, and is either *nb* or *nb*-1, or *n* if $n \leq nb$.

Note that U^H denotes the conjugate transpose of U.

This is an auxiliary routine called by ?hetrf_rook. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if *uplo* = 'U') or A_{22} (if *uplo* = 'L').

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:

= 'U': upper triangular

= 'L': lower triangular

n

INTEGER. The order of the matrix *A*. $n \geq 0$.

nb

INTEGER. The maximum number of columns of the matrix *A* that should be factored. *nb* should be at least 2 to allow for 2-by-2 pivot blocks.

a

COMPLEX for clahef_rook

DOUBLE COMPLEX for zlahef_rook.

Array, DIMENSION (*lda*, *n*).

On entry, the Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda

INTEGER. The leading dimension of the array *a*. $lda \geq \max(1,n)$.

w

COMPLEX for clahef_rook

DOUBLE COMPLEX for zlahef_rook.

Workspace array, DIMENSION (*ldw*, *nb*).

ldw

INTEGER. The leading dimension of the array *w*. $ldw \geq \max(1,n)$.

Output Parameters

kb

INTEGER. The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if $n \leq nb$.

a

On exit, *A* contains details of the partial factorization.

ipiv

INTEGER.

Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*.

If *uplo* = 'U', only the last *kb* elements of *ipiv* are set;

if *uplo* = 'L', only the first *kb* elements are set.

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) are interchanged and *D*(*k*, *k*) is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) < 0 and *ipiv*(*k*-1) < 0, then rows and columns *k* and -*ipiv*(*k*) are interchanged, rows and columns *k* - 1 and -*ipiv*(*k* - 1) are interchanged, and $D_{k-1:k,\ k-1:k}$ is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) < 0 and *ipiv*(*k* + 1) < 0, then rows and columns *k* and -*ipiv*(*k*) are interchanged, rows and columns *k* + 1 and -*ipiv*(*k* + 1) are interchanged, and $D_{k:k+1,\ k:k+1}$ is a 2-by-2 diagonal block.

info INTEGER.

= 0: successful exit

> 0: if *info* = *k*, *D*(*k*, *k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?latbs

Solves a triangular banded system of equations.

Syntax

```
call slatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
call dlatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
call clatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
call zlatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
```

Include Files

- mkl.fi

Description

The routine solves one of the triangular systems

*A*x = s*b*, or A^T*x = s*b*, or A^H*x = s*b* (for complex flavors)

with scaling to prevent overflow, where *A* is an upper or lower triangular band matrix. Here A^T denotes the transpose of *A*, A^H denotes the conjugate transpose of *A*, *x* and *b* are *n*-element vectors, and *s* is a scaling factor, usually less than or equal to 1, chosen so that the components of *x* will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine ?tbsv is called. If the matrix *A* is singular (*A*(*j*, *j*)=0 for some *j*), then *s* is set to 0 and a non-trivial solution to *A*x = 0 is returned.

Input Parameters

uplo CHARACTER*1.

Specifies whether the matrix *A* is upper or lower triangular.

= 'U': upper triangular

= 'L': lower triangular

trans CHARACTER*1.

Specifies the operation applied to *A*.

= 'N': solve $A*x = s*b$ (no transpose)

= 'T': solve $A^T*x = s*b$ (transpose)

= 'C': solve $A^H*x = s*b$ (conjugate transpose)

diag CHARACTER*1.

Specifies whether the matrix *A* is unit triangular

= 'N': non-unit triangular

= 'U': unit triangular

normin CHARACTER*1.

Specifies whether *cnorm* is set.

= 'Y': *cnorm* contains the column norms on entry;

= 'N': *cnorm* is not set on entry. On exit, the norms is computed and stored in *cnorm*.

n INTEGER. The order of the matrix *A*. $n \geq 0$.

kd INTEGER. The number of subdiagonals or superdiagonals in the triangular matrix *A*. $kb \geq 0$.

ab REAL for slatbs

DOUBLE PRECISION for dlatbs

COMPLEX for clatbs

DOUBLE COMPLEX for zlatbs.

Array, DIMENSION (*ldab*, *n*).

The upper or lower triangular band matrix *A*, stored in the first *kb*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

if *uplo* = 'U', $ab(kd+1+i-j,j) = A(i,j)$ for $\max(1, j-kd) \leq i \leq j$;

if *uplo* = 'L', $ab(1+i-j,j) = A(i,j)$ for $j \leq i \leq \min(n, j+kd)$.

ldab INTEGER. The leading dimension of the array *ab*. $ldab \geq kb+1$.

x REAL for slatbs

DOUBLE PRECISION for dlatbs

COMPLEX for clatbs

DOUBLE COMPLEX for zlatbs.

Array, DIMENSION (*n*).

On entry, the right hand side *b* of the triangular system.

cnorm REAL for slatbs/clatbs

DOUBLE PRECISION for dlatbs/zlatbs.

Array, DIMENSION (*n*).

If *NORMIN* = 'Y', *cnorm* is an input argument and *cnorm*(*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*.

If *trans* = 'N', *cnorm*(*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

Output Parameters

scale

REAL for slatbs/clatbs

DOUBLE PRECISION for dlatbs/zlatbs.

The scaling factor s for the triangular system as described above. If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to *Ax* = 0.

cnorm

If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

info

INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument had an illegal value

?latm1

Computes the entries of a matrix as specified.

Syntax

```
call slatm1( mode, cond, irsign, idist, iseed, d, n, info )
call dlatm1( mode, cond, irsign, idist, iseed, d, n, info )
call clatm1( mode, cond, irsign, idist, iseed, d, n, info )
call zlatm1( mode, cond, irsign, idist, iseed, d, n, info )
```

Include Files

- mkl.fi

Description

The ?latm1 routine computes the entries of *D*(1..*n*) as specified by *mode*, *cond* and *irsign*. *idist* and *iseed* determine the generation of random numbers.

?latm1 is called by slatmr (for slatm1 and dlatm1), and by clatmr(for clatm1 and zlatm1) to generate random test matrices for LAPACK programs.

Input Parameters

mode

INTEGER. On entry describes how *d* is to be computed:

mode = 0 means do not change *d*.

mode = 1 sets *d*(1) = 1 and *d*(2:*n*) = 1.0/*cond*

mode = 2 sets *d*(1:*n*-1) = 1 and *d*(*n*)=1.0/*cond*

mode = 3 sets *d*(*i*)=*cond***(-(*i*-1)/(*n*-1))

mode = 4 sets *d*(*i*)= 1 - (*i*-1)/(*n*-1)*(1 - 1/*cond*)

mode = 5 sets *d* to random numbers in the range (1/*cond* , 1) such that their logarithms are uniformly distributed.

mode = 6 sets *d* to random numbers from same distribution as the rest of the matrix.

mode < 0 has the same meaning as abs(*mode*), except that the order of the elements of *d* is reversed.

Thus if *mode* is positive, *d* has entries ranging from 1 to 1/*cond*, if negative, from 1/*cond* to 1.

cond

REAL for slatm1,

DOUBLE PRECISION for dlatm1,

REAL for clatm1,

DOUBLE PRECISION for zlatm1,

On entry, used as described under *mode* above. If used, it must be ≥ 1.

irsign

INTEGER.

On entry, if *mode* is not -6, 0, or 6, determines sign of entries of *d*.

If *irsign* = 0, entries of *d* are unchanged.

If *irsign* = 1, each entry of *d* is multiplied by a random complex number uniformly distributed with absolute value 1.

idist

INTEGER. Specifies the distribution of the random numbers.

For slatm1 and dlatm1:

= 1: uniform (0,1)

= 2: uniform (-1,1)

= 3: normal (0,1)

For clatm1 and zlatm1:

= 1: real and imaginary parts each uniform (0,1)

= 2: real and imaginary parts each uniform (-1,1)

= 3: real and imaginary parts each normal (0,1)

= 4: complex number uniform in disk(0, 1)

iseed

INTEGER. Array, size (4).

Specifies the seed of the random number generator. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers. The values of *iseed*(4) are changed on exit, and can be used in the next call to ?latm1 to continue the same random number sequence.

d

REAL for slatm1,

DOUBLE PRECISION for dlatm1,

	COMPLEX for clatm1, DOUBLE COMPLEX for zlatm1, Array, size *n*.
n	INTEGER. Number of entries of *d*.

Output Parameters

iseed	On exit, the seed is updated.
d	On exit, *d* is updated, unless *mode* = 0.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -1, *mode* is not in range -6 to 6. If *info* = -2, *mode* is neither -6, 0 nor 6, and *irsign* is neither 0 nor 1. If *info* = -3, *mode* is neither -6, 0 nor 6 and *cond* is less than 1. If *info* = -4, *mode* equals 6 or -6 and *idist* is not in range 1 to 4. If *info* = -7, *n* is negative.

?latm2

Returns an entry of a random matrix.

Syntax

```
res = slatm2( m, n, i, j, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
res = dlatm2( m, n, i, j, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
res = clatm2( m, n, i, j, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
res = zlatm2( m, n, i, j, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
```

Include Files

- mkl.fi

Description

The ?latm2 routine returns entry (*i* , *j*) of a random matrix of dimension (*m*, *n*). It is called by the ?latmr routine in order to build random test matrices. No error checking on parameters is done, because this routine is called in a tight loop by ?latmr which has already checked the parameters.

Use of ?latm2 differs from ?latm3 in the order in which the random number generator is called to fill in random matrix entries. With ?latm2, the generator is called to fill in the pivoted matrix columnwise. With ?latm2, the generator is called to fill in the matrix columnwise, after which it is pivoted. Thus, ?latm3

can be used to construct random matrices which differ only in their order of rows and/or columns. ?latm2 is used to construct band matrices while avoiding calling the random number generator for entries outside the band (and therefore generating random numbers).

The matrix whose (*i* , *j*) entry is returned is constructed as follows (this routine only computes one entry):

- If *i* is outside (1..*m*) or *j* is outside (1..n), returns zero (this is convenient for generating matrices in band format).
- Generate a matrix *A* with random entries of distribution *idist*.
- Set the diagonal to *D*.
- Grade the matrix, if desired, from the left (by *dl*) and/or from the right (by *dr* or *dl*) as specified by *igrade*.
- Permute, if desired, the rows and/or columns as specified by *ipvtng* and *iwork*.
- Band the matrix to have lower bandwidth *kl* and upper bandwidth *ku*.
- Set random entries to zero as specified by *sparse*.

Input Parameters

m — INTEGER. Number of rows of the matrix.

n — INTEGER. Number of columns of the matrix.

i — INTEGER. Row of the entry to be returned.

j — INTEGER. Column of the entry to be returned.

kl — INTEGER. Lower bandwidth.

ku — INTEGER. Upper bandwidth.

idist — INTEGER. On entry, *idist* specifies the type of distribution to be used to generate a random matrix .

for slatm2 and dlatm2:

= 1: uniform (0,1)

= 2: uniform (-1,1)

= 3: normal (0,1)

for clatm2 and zlatm2:

= 1: real and imaginary parts each uniform (0,1)

= 2: real and imaginary parts each uniform (-1,1)

= 3: real and imaginary parts each normal (0,1)

= 4: complex number uniform in disk (0, 1)

iseed — INTEGER. Array, size 4.

Seed for the random number generator.

d — REAL for slatm2,

DOUBLE PRECISION for dlatm2,

COMPLEX for clatm2,

DOUBLE COMPLEX for zlatm2,

Array, size (min(*i*, *j*)). Diagonal entries of matrix.

igrade INTEGER. Specifies grading of matrix as follows:

= 0: no grading

= 1: matrix premultiplied by diag(*dl*)

= 2: matrix postmultiplied by diag(*dr*)

= 3: matrix premultiplied by diag(*dl*) and postmultiplied by diag(*dr*)

= 4: matrix premultiplied by diag(*dl*) and postmultiplied by inv(diag(*dl*))

For `slatm2` and `slatm2`:

= 5: matrix premultiplied by diag(*dl*) and postmultiplied by diag(*dl*)

For `clatm2` and `zlatm2`:

= 5: matrix premultiplied by diag(*dl*) and postmultiplied by diag(conjg(*dl*))

= 6: matrix premultiplied by diag(*dl*) and postmultiplied by diag(*dl*)

dl `REAL` for `slatm2`,

`DOUBLE PRECISION` for `dlatm2`,

`COMPLEX` for `clatm2`,

`DOUBLE COMPLEX` for `zlatm2`,

Array, size (*i* or *j*), as appropriate.

Left scale factors for grading matrix.

dr `REAL` for `slatm2`,

`DOUBLE PRECISION` for `dlatm2`,

`COMPLEX` for `clatm2`,

`DOUBLE COMPLEX` for `zlatm2`,

Array, size (*i* or *j*), as appropriate.

Right scale factors for grading matrix.

ipvtng INTEGER. On entry specifies pivoting permutations as follows:

= 0: none

= 1: row pivoting

= 2: column pivoting

= 3: full pivoting, i.e., on both sides

iwork INTEGER.

Array, size (*i* or *j*), as appropriate. This array specifies the permutation used. The row (or column) in position *k* was originally in position *iwork*`( k )`. This differs from *iwork* for `?latm3`.

sparse `REAL` for `slatm2`,

`DOUBLE PRECISION` for `dlatm2`,

`REAL` for `clatm2`,

DOUBLE PRECISION for zlatm2,

Specifies the sparsity of the matrix. If sparse matrix is to be generated, *sparse* should lie between 0 and 1. A uniform (0, 1) random number *x* is generated and compared to *sparse*. If *x* is larger the matrix entry is unchanged and if *x* is smaller the entry is set to zero. Thus on the average a fraction *sparse* of the entries will be set to zero.

Output Parameters

iseed INTEGER.

On exit, the seed is updated.

res REAL for slatm2,

DOUBLE PRECISION for dlatm2,

COMPLEX for clatm2,

DOUBLE COMPLEX for zlatm2,

Entry of a random matrix.

?latm3

Returns set entry of a random matrix.

Syntax

```
res = slatm3( m, n, i, j, isub, jsub, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
res = dlatm3( m, n, i, j, isub, jsub, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
res = clatm3( m, n, i, j, isub, jsub, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
res = zlatm3( m, n, i, j, isub, jsub, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
```

Include Files

- mkl.fi

Description

The ?latm3 routine returns the (*isub*, *jsub*) entry of a random matrix of dimension (*m*, *n*) described by the other parameters. (*isub*, *jsub*) is the final position of the (*i* ,*j*) entry after pivoting according to *ipvtng* and *iwork*. ?latm3 is called by the ?latmr routine in order to build random test matrices. No error checking on parameters is done, because this routine is called in a tight loop by ?latmr which has already checked the parameters.

Use of ?latm3 differs from ?latm2 in the order in which the random number generator is called to fill in random matrix entries. With ?latm2, the generator is called to fill in the pivoted matrix columnwise. With ?latm3, the generator is called to fill in the matrix columnwise, after which it is pivoted. Thus, ?latm3 can be used to construct random matrices which differ only in their order of rows and/or columns. ?latm2 is used to construct band matrices while avoiding calling the random number generator for entries outside the band (and therefore generating random numbers in different orders for different pivot orders).

The matrix whose (*isub*, *jsub*) entry is returned is constructed as follows (this routine only computes one entry):

- If *isub* is outside `(1..m)` or *jsub* is outside `(1..n)`, returns zero (this is convenient for generating matrices in band format).
- Generate a matrix *A* with random entries of distribution *idist*.
- Set the diagonal to *D*.
- Grade the matrix, if desired, from the left (by *dl*) and/or from the right (by *dr* or *dl*) as specified by *igrade*.
- Permute, if desired, the rows and/or columns as specified by *ipvtng* and *iwork*.
- Band the matrix to have lower bandwidth *kl* and upper bandwidth *ku*.
- Set random entries to zero as specified by *sparse*.

Input Parameters

m — `INTEGER`. Number of rows of matrix.

n — `INTEGER`. Number of columns of matrix.

i — `INTEGER`. Row of unpivoted entry to be returned.

j — `INTEGER`. Column of unpivoted entry to be returned.

isub — `INTEGER`. Row of pivoted entry to be returned.

jsub — `INTEGER`. Column of pivoted entry to be returned.

kl — `INTEGER`. Lower bandwidth.

ku — `INTEGER`. Upper bandwidth.

idist — `INTEGER`. On entry, *idist* specifies the type of distribution to be used to generate a random matrix.

for `slatm2` and `dlatm2`:

= 1: uniform (0,1)

= 2: uniform (-1,1)

= 3: normal (0,1)

for `clatm2` and `zlatm2`:

= 1: real and imaginary parts each uniform (0,1)

= 2: real and imaginary parts each uniform (-1,1)

= 3: real and imaginary parts each normal (0,1)

= 4: complex number uniform in disk(0, 1)

iseed — `INTEGER`. Array, size 4.

Seed for random number generator.

d — `REAL` for `slatm3`,

`DOUBLE PRECISION` for `dlatm3`,

`COMPLEX` for `clatm3`,

`DOUBLE COMPLEX` for `zlatm3`,

Array, size `(min(i, j))`. Diagonal entries of matrix.

igrade INTEGER. Specifies grading of matrix as follows:

= 0: no grading

= 1: matrix premultiplied by diag(*dl*)

= 2: matrix postmultiplied by diag(*dr*)

= 3: matrix premultiplied by diag(*dl*) and postmultiplied by diag(*dr*)

= 4: matrix premultiplied by diag(*dl*) and postmultiplied by inv(diag(*dl*))

For slatm2 and slatm2:

= 5: matrix premultiplied by diag(*dl*) and postmultiplied by diag(*dl*)

For clatm2 and zlatm2:

= 5: matrix premultiplied by diag(*dl*) and postmultiplied by diag(conjg(*dl*))

= 6: matrix premultiplied by diag(*dl*) and postmultiplied by diag(*dl*)

dl REAL for slatm3,

DOUBLE PRECISION for dlatm3,

COMPLEX for clatm3,

DOUBLE COMPLEX for zlatm3,

Array, size (*i* or *j*, as appropriate).

Left scale factors for grading matrix.

dr REAL for slatm3,

DOUBLE PRECISION for dlatm3,

COMPLEX for clatm3,

DOUBLE COMPLEX for zlatm3,

Array, size (*i* or *j*, as appropriate).

Right scale factors for grading matrix.

ipvtng INTEGER. On entry specifies pivoting permutations as follows:

If *ipvtng* = 0: none.

If *ipvtng* = 1: row pivoting.

If *ipvtng* = 2: column pivoting.

If *ipvtng* = 3: full pivoting, i.e., on both sides.

sparse REAL for slatm3,

DOUBLE PRECISION for dlatm3,

REAL for clatm3,

DOUBLE PRECISION for zlatm3,

	On entry, specifies the sparsity of the matrix if sparse matrix is to be generated. *sparse* should lie between 0 and 1. A uniform(0, 1) random number *x* is generated and compared to *sparse*; if *x* is larger the matrix entry is unchanged and if *x* is smaller the entry is set to zero. Thus on the average a fraction *sparse* of the entries will be set to zero.
iwork	INTEGER. Array, size (*i* or *j*, as appropriate). This array specifies the permutation used. The row (or column) originally in position *k* is in position *iwork(k)* after pivoting. This differs from *iwork* for ?latm2.

Output Parameters

isub	On exit, row of pivoted entry is updated.
jsub	On exit, column of pivoted entry is updated.
iseed	On exit, the seed is updated.
res	REAL for slatm3, DOUBLE PRECISION for dlatm3, COMPLEX for clatm3, DOUBLE COMPLEX for zlatm3, Entry of a random matrix.

?latm5

Generates matrices involved in the Generalized Sylvester equation.

Syntax

```
call slatm5( prtype, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, r, ldr, l,
ldl, alpha, qblcka, qblckb )
call dlatm5( prtype, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, r, ldr, l,
ldl, alpha, qblcka, qblckb )
call clatm5( prtype, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, r, ldr, l,
ldl, alpha, qblcka, qblckb )
call zlatm5( prtype, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, r, ldr, l,
ldl, alpha, qblcka, qblckb )
```

Include Files

- mkl.fi

Description

The ?latm5 routine generates matrices involved in the Generalized Sylvester equation:

*A * R - L * B = C*

*D * R - L * E = F*

They also satisfy the diagonalization condition:

$$\begin{bmatrix} I & -L \\ & I \end{bmatrix}\begin{bmatrix} A & -C \\ & B \end{bmatrix}\begin{bmatrix} I & R \\ & I \end{bmatrix}=\begin{bmatrix} A & \\ & B \end{bmatrix}$$
$$\begin{bmatrix} I & -L \\ & I \end{bmatrix}\begin{bmatrix} D & -F \\ & E \end{bmatrix}\begin{bmatrix} I & R \\ & I \end{bmatrix}=\begin{bmatrix} D & \\ & E \end{bmatrix}$$

Input Parameters

prtype

INTEGER. Specifies the type of matrices to generate.

- If *prtype* = 1, *A* and *B* are Jordan blocks, *D* and *E* are identity matrices.

  A:

  If (*i* == *j*) then $A_{i,j}$ = 1.0.

  If (*j* == *i* + 1) then $A_{i,j}$ = -1.0.

  Otherwise $A_{i,j}$ = 0.0, *i*, *j* = 1...*m*

  B:

  If (*i* == *j*) then $B_{i,j}$ = 1.0 - *alpha*.

  If (*j* == *i* + 1) then $B_{i,j}$ = 1.0 .

  Otherwise $B_{i,j}$ = 0.0, *i*, *j* = 1...*n*.

  D:

  If (*i* == *j*) then $D_{i,j}$ = 1.0.

  Otherwise $D_{i,j}$ = 0.0, *i*, *j* = 1...*m*.

  E:

  If (*i* == *j*) then $E_{i,j}$ = 1.0

  Otherwise $E_{i,j}$ = 0.0, *i*, *j* = 1...*n*.

  L = *R* are chosen from [-10...10], which specifies the right hand sides (*C*, *F*).
- If *prtype* = 2 or 3: Triangular and/or quasi- triangular.

  A:

  If (*i* ≤ *j*) then $A_{i,j}$ = [-1...1].

  Otherwise $A_{i,j}$ = 0.0, *i*, *j* = 1...M.

  If (*prtype* = 3) then $A_{k+1,k+1} = A_{k,k}$;

  $A_{k+1,k}$ = [-1...1];

  sign($A_{k,k+1}$) = -(sign($A_{k+1,k}$).

  k = 1, *m*- 1, *qblcka*

  B :

  If (*i* ≤ *j*) then $B_{i,j}$ = [-1...1].

  Otherwise $B_{i,j}$ = 0.0, *i*, *j* = 1...*n*.

  If (*prtype* = 3) then$B_{k+1,k+1} = B_{k,k}$

$B_{k+1,k}$ = [-1...1]

sign($B_{k,k+1}$)= -(sign($B_{k+1,k}$)

k = 1, n - 1, $qblckb$.

D:

If ($i \leq j$) then $D_{i,j}$ = [-1...1].

Otherwise $D_{i,j}$ = 0.0, i, j = 1...m.

E:

If (i <= j) then $E_{i,j}$ = [-1...1].

Otherwise $E_{i,j}$ = 0.0, i, j = 1...N.

L, R are chosen from [-10...10], which specifies the right hand sides (C, F).

- If $prtype$ = 4 Full

  $A_{i,j}$ = [-10...10]

  $D_{i,j}$ = [-1...1] i,j = 1...m

  $B_{i,j}$ = [-10...10]

  $E_{i,j}$ = [-1...1] i,j = 1...n

  $R_{i,j}$ = [-10...10]

  $L_{i,j}$ = [-1...1] i = 1..m ,j = 1...n

  L and R specifies the right hand sides (C, F).
- If $prtype$ = 5 special case common and/or close eigs.

m — INTEGER. Specifies the order of A and D and the number of rows in C, F, R and L.

n — INTEGER. Specifies the order of B and E and the number of columns in C, F, R and L.

lda — INTEGER. The leading dimension of a.

ldb — INTEGER. The leading dimension of b.

ldc — INTEGER. The leading dimension of c.

ldd — INTEGER. The leading dimension of d.

lde — INTEGER. The leading dimension of e.

ldf — INTEGER. The leading dimension of f.

ldr — INTEGER. The leading dimension of r.

ldl — INTEGER. The leading dimension of l.

alpha — REAL for slatm5,

DOUBLE PRECISION for dlatm5,

REAL for clatm5,

DOUBLE PRECISION for zlatm5,

Parameter used in generating *prtype* = 1 and 5 matrices.

qblcka — INTEGER. When *prtype* = 3, specifies the distance between 2-by-2 blocks on the diagonal in *A*. Otherwise, *qblcka* is not referenced. *qblcka* > 1.

qblckb — INTEGER. When *prtype* = 3, specifies the distance between 2-by-2 blocks on the diagonal in *B*. Otherwise, *qblckb* is not referenced. *qblckb* > 1.

Output Parameters

a

REAL for slatm5,

DOUBLE PRECISION for dlatm5,

COMPLEX for clatm5,

DOUBLE COMPLEX for zlatm5,

Array, size (*lda*, *m*). On exit *a* contains the*m*-by-*m* array *A* initialized according to *prtype*.

b

REAL for slatm5,

DOUBLE PRECISION for dlatm5,

COMPLEX for clatm5,

DOUBLE COMPLEX for zlatm5,

Array, size (*ldb*, *n*). On exit *b* contains the *n*-by-*n* array *B* initialized according to *prtype*.

c

REAL for slatm5,

DOUBLE PRECISION for dlatm5,

COMPLEX for clatm5,

DOUBLE COMPLEX for zlatm5,

Array, size (*ldc*, *n*). On exit *c* contains the *m*-by-*n* array *C* initialized according to *prtype*.

d

REAL for slatm5,

DOUBLE PRECISION for dlatm5,

COMPLEX for clatm5,

DOUBLE COMPLEX for zlatm5,

Array, size (*ldd*, *m*). On exit *d* contains the *m*-by-*m* array *D* initialized according to *prtype*.

e

REAL for slatm5,

DOUBLE PRECISION for dlatm5,

COMPLEX for clatm5,

DOUBLE COMPLEX for zlatm5,

Array, size (*lde*, *n*). On exit *e* contains the *n*-by-*n* array *E* initialized according to *prtype*.

f	REAL for slatm5, DOUBLE PRECISION for dlatm5, COMPLEX for clatm5, DOUBLE COMPLEX for zlatm5, Array, size (*ldf*, *n*). On exit *f* contains the *m*-by-*n* array *F* initialized according to *prtype*.
r	REAL for slatm5, DOUBLE PRECISION for dlatm5, COMPLEX for clatm5, DOUBLE COMPLEX for zlatm5, Array, size (*ldr*, *n*). On exit *R* contains the *m*-by-*n* array *R* initialized according to *prtype*.
l	REAL for slatm5, DOUBLE PRECISION for dlatm5, COMPLEX for clatm5, DOUBLE COMPLEX for zlatm5, Array, size (*ldl*, *n*). On exit *l* contains the *m*-by-*n*array *L* initialized according to *prtype*.

?latm6

Generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.

Syntax

```
call slatm6( type, n, a, lda, b, x, ldx, y, ldy, alpha, beta, wx, wy, s, dif )
call dlatm6( type, n, a, lda, b, x, ldx, y, ldy, alpha, beta, wx, wy, s, dif )
call clatm6( type, n, a, lda, b, x, ldx, y, ldy, alpha, beta, wx, wy, s, dif )
call zlatm6( type, n, a, lda, b, x, ldx, y, ldy, alpha, beta, wx, wy, s, dif )
```

Include Files

- mkl.fi

Description

The ?latm6 routine generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.

There two kinds of test matrix pairs:

(*A*, *B*)= inverse(*YH*) * (*Da*, *Db*) * inverse(*X*)

Type 1:

$$Da = \begin{bmatrix} 1+a & 0 & 0 & 0 & 0 \\ 0 & 2+a & 0 & 0 & 0 \\ 0 & 0 & 3+a & 0 & 0 \\ 0 & 0 & 0 & 4+a & 0 \\ 0 & 0 & 0 & 0 & 5+2 \end{bmatrix} \quad Db = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Type 2:

$$Da = \begin{bmatrix} 1+i & 0 & 0 & 0 & 0 \\ 0 & 1-i & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & (1+a)+(1+b)i & 0 \\ 0 & 0 & 0 & 0 & (1+a)-(1+b)i \end{bmatrix} \quad Db = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

In both cases the same inverse(*YH*) and inverse(X) are used to compute (*A*, *B*), giving the exact eigenvectors to (*A*,*B*) as (*YH*, *X*):

$$YH = \begin{bmatrix} 1 & 0 & -y & y & -y \\ 0 & 1 & -y & y & -y \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad X = \begin{bmatrix} 1 & 0 & -x & -x & x \\ 0 & 1 & x & -x & -x \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

,

where *a*, *b*, *x* and *y* will have all values independently of each other.

Input Parameters

type	INTEGER. Specifies the problem type.
n	INTEGER. Size of the matrices *A* and *B*.
lda	INTEGER. The leading dimension of *a* and of *b*.
ldx	INTEGER. The leading dimension of *x*.
ldy	INTEGER. The leading dimension of *y*.
alpha, *beta*	REAL for slatm6, DOUBLE PRECISION for dlatm6, COMPLEX for clatm6, DOUBLE COMPLEX for zlatm6, Weighting constants for matrix *A*.
wx	REAL for slatm6,

DOUBLE PRECISION for dlatm6,

COMPLEX for clatm6,

DOUBLE COMPLEX for zlatm6,

Constant for right eigenvector matrix.

wy REAL for slatm6,

DOUBLE PRECISION for dlatm6,

COMPLEX for clatm6,

DOUBLE COMPLEX for zlatm6,

Constant for left eigenvector matrix.

Output Parameters

a REAL for slatm6,

DOUBLE PRECISION for dlatm6,

COMPLEX for clatm6,

DOUBLE COMPLEX for zlatm6,

Array, size (*lda*, *n*). On exit, *a* contains the *n*-by-*n* matrix initialized according to *type*.

b REAL for slatm26,

DOUBLE PRECISION for dlatm6,

COMPLEX for clatm6,

DOUBLE COMPLEX for zlatm6,

Array, size (*lda*, *n*). On exit, *b* contains the *n*-by-*n* matrix initialized according to *type*.

x REAL for slatm6,

DOUBLE PRECISION for dlatm6,

COMPLEX for clatm6,

DOUBLE COMPLEX for zlatm6,

Array, size (*ldx*, *n*). On exit, *x* contains the *n*-by-*n* matrix of right eigenvectors.

y REAL for slatm6,

DOUBLE PRECISION for dlatm6,

COMPLEX for clatm6,

DOUBLE COMPLEX for zlatm6,

Array, size (*ldy*, *n*). On exit, *y* is the *n*-by-*n* matrix of left eigenvectors.

s REAL for slatm6,

DOUBLE PRECISION for dlatm6,

REAL for clatm6,

DOUBLE PRECISION for zlatm6,

Array, size (*n*). *s*(*i*) is the reciprocal condition number for eigenvalue *i* .

dif

REAL for slatm6,

DOUBLE PRECISION for dlatm6,

REAL for clatm6,

DOUBLE PRECISION for zlatm6,

Array, size(*n*). *dif*(*i*) is the reciprocal condition number for eigenvector *i* .

?latme

Generates random non-symmetric square matrices with specified eigenvalues.

Syntax

```
call slatme( n, dist, iseed, d, mode, cond, dmax, ei, rsign, upper, sim, ds, modes, conds, kl, ku, anorm, a, lda, work, info )
call dlatme( n, dist, iseed, d, mode, cond, dmax, ei, rsign, upper, sim, ds, modes, conds, kl, ku, anorm, a, lda, work, info )
call clatme( n, dist, iseed, d, mode, cond, dmax, ei, rsign, upper, sim, ds, modes, conds, kl, ku, anorm, a, lda, work, info )
call zlatme( n, dist, iseed, d, mode, cond, dmax, ei, rsign, upper, sim, ds, modes, conds, kl, ku, anorm, a, lda, work, info )
```

Include Files

- mkl.fi

Description

The ?latme routine generates random non-symmetric square matrices with specified eigenvalues. ?latme operates by applying the following sequence of operations:

1. Set the diagonal to *d*, where *d* may be input or computed according to *mode*, *cond*, *dmax*, and *rsign* as described below.
2. If *upper* = 'T', the upper triangle of *a* is set to random values out of distribution *dist*.
3. If *sim*='T', *a* is multiplied on the left by a random matrix *X*, whose singular values are specified by *ds*, *modes*, and *conds*, and on the right by *X* inverse.
4. If *kl* < *n*-1, the lower bandwidth is reduced to *kl* using Householder transformations. If *ku* < *n*-1, the upper bandwidth is reduced to *ku*.
5. If *anorm* is not negative, the matrix is scaled to have maximum-element-norm *anorm*.

NOTE
Since the matrix cannot be reduced beyond Hessenberg form, no packing options are available.

Input Parameters

n

INTEGER. The number of columns (or rows) of *A*.

dist

CHARACTER*1. On entry, *dist* specifies the type of distribution to be used to generate the random eigen-/singular values, and on the upper triangle (see *upper*).

If *dist* = 'U': uniform(0, 1)

If *dist* = 'S': uniform(-1, 1)

If *dist* = 'N': normal(0, 1)

If *dist* = 'D': uniform on the complex disc |*z*| < 1.

iseed

INTEGER. Array, size 4.

On entry *iseed* specifies the seed of the random number generator. The elements should lie between 0 and 4095 inclusive, and *iseed*(4) should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers.

d

REAL for slatme,

DOUBLE PRECISION for dlatme,

COMPLEX for clatme,

DOUBLE COMPLEX for zlatme,

Array, size (*n*). This array is used to specify the eigenvalues of *A*.

If *mode* = 0, then *d* is assumed to contain the eigenvalues. Otherwise they are computed according to *mode*, *cond*, *dmax*, and *rsign* and placed in *d*.

mode

INTEGER. On entry *mode* describes how the eigenvalues are to be specified:

mode = 0 means use *d* (with *ei* for slatme and dlatme) as input.

mode = 1 sets *d*(1) = 1 and *d*[1:*n* - 1]=1.0/*cond*.

mode = 2 sets *d*(1:*n*-1) = 1 and *d*(*n*)=1.0/*cond*.

mode = 3 sets *d*(*i*) = *cond***(-(*i*-1)/(*n*-1)).

mode = 4 sets *d*(*i*) = 1 - (*i*-1)/(*n*-1)*(1 - 1/*cond*).

mode = 5 sets *d* to random numbers in the range (1/*cond* , 1) such that their logarithms are uniformly distributed.

mode = 6 sets *d* to random numbers from same distribution as the rest of the matrix.

mode < 0 has the same meaning as abs(*mode*), except that the order of the elements of *d* is reversed.

Thus if *mode* is between 1 and 4, *d* has entries ranging from 1 to 1/*cond*, if between -1 and -4, *d* has entries ranging from 1/*cond* to 1.

cond

REAL for slatme,

DOUBLE PRECISION for dlatme,

REAL for clatme,

DOUBLE PRECISION for zlatme,

On entry, this is used as described under *mode* above. If used, it must be ≥ 1.

dmax

REAL for slatme,

DOUBLE PRECISION for dlatme,

COMPLEX for clatme,

DOUBLE COMPLEX for zlatme,

If *mode* is not -6, 0 or 6, the contents of *d* as computed according to *mode* and *cond* are scaled by *dmax* / max(abs(*d*(*i*))). Note that *dmax* needs not be positive or real: if *dmax* is negative or complex (or zero), *d* will be scaled by a negative or complex number (or zero). If *rsign*='F' then the largest (absolute) eigenvalue will be equal to *dmax*.

ei

CHARACTER*1. Used by slatme and dlatme only.

Array, size (*n*).

If *mode* = 0, and *ei*(1) is not ' ' (space character), this array specifies which elements of *d* (on input) are real eigenvalues and which are the real and imaginary parts of a complex conjugate pair of eigenvalues. The elements of *ei* may then only have the values 'R' and 'I'.

If *ei*(*j*) = 'R' and *ei*(*j* + 1) = 'I', then the *j* -th eigenvalue is cmplx(*d*(*j*) , *d*(*j* + 1)), and the (*j* +1)-th is the complex conjugate thereof.

If *ei*(*j*) = *ei*(*j* + 1)='R', then the *j*-th eigenvalue is *d*(*j*) (i.e., real). *ei*(1) may not be 'I', nor may two adjacent elements of *ei* both have the value 'I'.

If *mode* is not 0, then *ei* is ignored. If *mode* is 0 and *ei*(1) = ' ', then the eigenvalues will all be real.

rsign

CHARACTER*1. If *mode* is not 0, 6, or -6, and *rsign* = 'T', then the elements of *d*, as computed according to *mode* and *cond*, are multiplied by a random sign (+1 or -1) for slatme and dlatme or by a complex number from the unit circle |*z*| = 1 for clatme and zlatme.

If *rsign* = 'F', the elements of *d* are not multiplied. *rsign* may only have the values 'T' or 'F'.

upper

CHARACTER*1. If *upper* = 'T', then the elements of *a* above the diagonal will be set to random numbers out of *dist*.

If *upper* = 'F', they will not. *upper* may only have the values 'T' or 'F'.

sim

CHARACTER*1. If *sim* = 'T', then *a* will be operated on by a "similarity transform", i.e., multiplied on the left by a matrix *X* and on the right by *X* inverse. *X* = *USV*, where *U* and *V* are random unitary matrices and *S* is a (diagonal) matrix of singular values specified by *ds*, *modes*, and *conds*.

If *sim* = 'F', then *a* will not be transformed.

ds

REAL for slatme,

DOUBLE PRECISION for dlatme,

REAL for clatme,

DOUBLE PRECISION for zlatme,

This array is used to specify the singular values of *X*, in the same way that *d* specifies the eigenvalues of *a*. If *mode* = 0, the *ds* contains the singular values, which may not be zero.

modes INTEGER.

Similar to *mode*, but for specifying the diagonal of *S*. *modes* = -6 and +6 are not allowed (since they would result in randomly ill-conditioned eigenvalues.)

conds REAL for slatme,

DOUBLE PRECISION for dlatme,

REAL for clatme,

DOUBLE PRECISION for zlatme,

Similar to *cond*, but for specifying the diagonal of *S*.

kl INTEGER. This specifies the lower bandwidth of the matrix. *kl* = 1 specifies upper Hessenberg form. If *kl* is at least *n*-1, then *A* will have full lower bandwidth.

ku INTEGER. This specifies the upper bandwidth of the matrix. *ku* = 1 specifies lower Hessenberg form.

If *ku* is at least *n*-1, then *a* will have full upper bandwidth.

If *ku* and *ku* are both at least *n*-1, then *a* will be dense. Only one of *ku* and *kl* may be less than *n*-1.

anorm REAL for slatme,

DOUBLE PRECISION for dlatme,

REAL for clatme,

DOUBLE PRECISION for zlatme,

If *anorm* is not negative, then *a* is scaled by a non-negative real number to make the maximum-element-norm of *a* to be *anorm*.

lda INTEGER. Number of rows of matrix *A*.

work REAL for slatme,

DOUBLE PRECISION for dlatme,

COMPLEX for clatme,

DOUBLE COMPLEX for zlatme,

Array, size (3*n) . Workspace.

Output Parameters

iseed INTEGER.

	On exit, the seed is updated.
d	Modified if *mode* is nonzero.
ds	Modified if *mode* is nonzero.
a	REAL for slatme, DOUBLE PRECISION for dlatme, COMPLEX for clatme, DOUBLE COMPLEX for zlatme, Array, size (*lda*, *n*). On exit, *a* is the desired test matrix.
info	INTEGER. If *info* = 0, execution is successful. If *info* = -1, *n* is negative . If *info* = -2, *dist* is an illegal string. If *info* = -5, *mode* is not in range -6 to 6. If *info* = -6, *cond* is less than 1.0, and *mode* is not -6, 0, or 6 . If *info* = -9, *rsign* is not 'T' or 'F' . If *info* = -10, *upper* is not 'T' or 'F'. If *info* = -11, *sim* is not 'T' or 'F'. If *info* = -12, *modes* = 0 and *ds* has a zero singular value. If *info* = -13, *modes* is not in the range -5 to 5. If *info* = -14, *modes* is nonzero and *conds* is less than 1. . If *info* = -15, *kl* is less than 1. If *info* = -16, *ku* is less than 1, or *kl* and *ku* are both less than *n*-1. If *info* = -19, *lda* is less than *m*. If *info* = 1, error return from ?latm1 (computing *d*) . If *info* = 2, cannot scale to *dmax* (max. eigenvalue is 0) . If *info* = 3, error return from slatm1(for slatme and clatme), dlatm1 (for dlatme and zlatme) . If *info* = 4, error return from ?large. If *info* = 5, zero singular value from slatm1(for slatme and clatme), dlatm1(for dlatme and zlatme).

?latmr

Generates random matrices of various types.

Syntax

```
call slatmr (m, n, dist, iseed, sym, d, mode, cond, dmax, rsign, grade, dl, model,
condl, dr, moder, condr, pivtng, ipivot, kl, ku, sparse, anorm, pack, a, lda, iwork,
info)
```

```
call dlatmr (m, n, dist, iseed, sym, d, mode, cond, dmax, rsign, grade, dl, model,
condl, dr, moder, condr, pivtng, ipivot, kl, ku, sparse, anorm, pack, a, lda, iwork,
info)
call clatmr (m, n, dist, iseed, sym, d, mode, cond, dmax, rsign, grade, dl, model,
condl, dr, moder, condr, pivtng, ipivot, kl, ku, sparse, anorm, pack, a, lda, iwork,
info)
call zlatmr (m, n, dist, iseed, sym, d, mode, cond, dmax, rsign, grade, dl, model,
condl, dr, moder, condr, pivtng, ipivot, kl, ku, sparse, anorm, pack, a, lda, iwork,
info)
```

Description

The ?latmr routine operates by applying the following sequence of operations:

1. Generate a matrix *A* with random entries of distribution *dist*:

   If *sym* = 'S', the matrix is symmetric,

   If *sym* = 'H', the matrix is Hermitian,

   If *sym* = 'N', the matrix is nonsymmetric.
2. Set the diagonal to *D*, where *D* may be input or computed according to *mode*, *cond*, *dmax* and *rsign* as described below.
3. Grade the matrix, if desired, from the left or right as specified by grade. The inputs *dl*, *model*, *condl*, *dr*, *moder* and *condr* also determine the grading as described below.
4. Permute, if desired, the rows and/or columns as specified by *pivtng* and *ipivot*.
5. Set random entries to zero, if desired, to get a random sparse matrix as specified by *sparse*.
6. Make *A* a band matrix, if desired, by zeroing out the matrix outside a band of lower bandwidth *kl* and upper bandwidth *ku*.
7. Scale *A*, if desired, to have maximum entry *anorm*.
8. Pack the matrix if desired. See options specified by the *pack* parameter.

> **NOTE**
> If two calls to ?latmr differ only in the pack parameter, they generate mathematically equivalent matrices. If two calls to ?latmr both have full bandwidth (*kl* = *m*-1 and *ku* = *n*-1), and differ only in the *pivtng* and *pack* parameters, then the matrices generated differ only in the order of the rows and columns, and otherwise contain the same data. This consistency cannot be and is not maintained with less than full bandwidth.

Input Parameters

m — INTEGER. Number of rows of *A*.

n — INTEGER. Number of columns of *A*.

dist — CHARACTER. On entry, *dist* specifies the type of distribution to be used to generate a random matrix .

If *dist* = 'U', real and imaginary parts are independent uniform(0, 1).

If *dist* = 'S', real and imaginary parts are independent uniform(-1, 1).

If *dist* = 'N', real and imaginary parts are independent normal(0, 1).

If *dist* = 'D', distribution is uniform on interior of unit disk.

iseed INTEGER. Array, size 4.

On entry, *iseed* specifies the seed of the random number generator. They should lie between 0 and 4095 inclusive, and *iseed*(4) should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers.

sym CHARACTER. If *sym* = 'S', generated matrix is symmetric.

If *sym* = 'H', generated matrix is Hermitian.

If *sym* = 'N', generated matrix is nonsymmetric.

d REAL for slatmr,

DOUBLE PRECISION for dlatmr,

COMPLEX for clatmr,

DOUBLE COMPLEX for zlatmr,

On entry this array specifies the diagonal entries of the diagonal of *A*. *d* may either be specified on entry, or set according to *mode* and *cond* as described below. If the matrix is Hermitian, the real part of *d* is taken. May be changed on exit if *mode* is nonzero.

mode INTEGER. On entry describes how *d* is to be used:

mode = 0 means use *d* as input.

mode = 1 sets *d*(1)=1 and *d*(2:*n*)=1.0/*cond*.

mode = 2 sets *d*(1:*n*-1)=1 and *d*(*n*)=1.0/*cond*.

mode = 3 sets *d*(*i*)=*cond***(-(*i*-1)/(*n*-1)).

mode = 4 sets *d*(*i*)=1 - (*i*-1)/(*n*-1)*(1 - 1/*cond*).

mode = 5 sets *d* to random numbers in the range (1/*cond* , 1) such that their logarithms are uniformly distributed.

mode = 6 sets *d* to random numbers from same distribution as the rest of the matrix.

mode < 0 has the same meaning as abs(*mode*), except that the order of the elements of *d* is reversed.

Thus if *mode* is between 1 and 4, *d* has entries ranging from 1 to 1/*cond*, if between -1 and -4, *D* has entries ranging from 1/*cond* to 1.

cond REAL for slatmr,

DOUBLE PRECISION for dlatmr,

REAL for clatmr,

DOUBLE PRECISION for zlatmr,

On entry, used as described under *mode* above. If used, *cond* must be ≥ 1.

dmax REAL for slatmr,

DOUBLE PRECISION for dlatmr,

COMPLEX for clatmr,

DOUBLE COMPLEX for zlatmr,

If *mode* is not -6, 0, or 6, the diagonal is scaled by *dmax* / max(abs(*d*(*i*))), so that maximum absolute entry of diagonal is abs(*dmax*). If *dmax* is complex (or zero), the diagonal is scaled by a complex number (or zero).

rsign

CHARACTER. If *mode* is not -6, 0, or 6, specifies the sign of the diagonal as follows:

For slatmr and dlatmr, if *rsign* = 'T', diagonal entries are multiplied 1 or -1 with a probability of 0.5.

For clatmr and zlatmr, if *rsign* = 'T', diagonal entries are multiplied by a random complex number uniformly distributed with absolute value 1.

If *rsign* = 'F', diagonal entries are unchanged.

grade

CHARACTER. Specifies grading of matrix as follows:

If *grade* = 'N', there is no grading

If *grade* = 'L', matrix is premultiplied by diag(*dl*) (only if matrix is nonsymmetric)

If *grade* = 'R', matrix is postmultiplied by diag(*dr*) (only if matrix is nonsymmetric)

If *grade* = 'B', matrix is premultiplied by diag(*dl*) and postmultiplied by diag(*dr*) (only if matrix is nonsymmetric)

If *grade* = 'H', matrix is premultiplied by diag(*dl*) and postmultiplied by diag(conjg(*dl*)) (only if matrix is Hermitian or nonsymmetric)

If *grade* = 'S', matrix is premultiplied by diag(*dl*) and postmultiplied by diag(*dl*) (only if matrix is symmetric or nonsymmetric)

If *grade* = 'E', matrix is premultiplied by diag(*dl*) and postmultiplied by inv(diag(*dl*)) (only if matrix is nonsymmetric)

NOTE
if *grade* = 'E', then *m* must equal *n*.

dl

REAL for slatmr,

DOUBLE PRECISION for dlatmr,

COMPLEX for clatmr,

DOUBLE COMPLEX for zlatmr,

Array, size (*m*).

If *model* = 0, then on entry this array specifies the diagonal entries of a diagonal matrix used as described under grade above.

If *model* is not zero, then *dl* is set according to *model* and *condl*, analogous to the way *D* is set according to *mode* and *cond* (except there is no *dmax* parameter for *dl*).

If *grade* = 'E', then *dl* cannot have zero entries.

Not referenced if *grade* = 'N' or 'R'. Changed on exit.

model INTEGER. This specifies how the diagonal array *dl* is computed, just as *mode* specifies how *D* is computed.

condl REAL for slatmr,

DOUBLE PRECISION for dlatmr,

REAL for clatmr,

DOUBLE PRECISION for zlatmr,

When *model* is not zero, this specifies the condition number of the computed *dl*.

dr REAL for slatmr,

DOUBLE PRECISION for dlatmr,

COMPLEX for clatmr,

DOUBLE COMPLEX for zlatmr,

If *moder* = 0, then on entry this array specifies the diagonal entries of a diagonal matrix used as described under *grade* above.

If *moder* is not zero, then *dr* is set according to *moder* and *condr*, analogous to the way *d* is set according to *mode* and *cond* (except there is no *dmax* parameter for *dr*).

Not referenced if *grade* = 'N', 'L', 'H''S' or 'E'.

moder INTEGER. This specifies how the diagonal array *dr* is to be computed, just as mode specifies how *d* is to be computed.

condr REAL for slatmr and clatmr,

DOUBLE PRECISION for dlatmr and zlatmr,

When *moder* is not zero, this specifies the condition number of the computed *dr*.

pivtng CHARACTER. On entry specifies pivoting permutations as follows:

If *pivtng* = 'N' or ' ': no pivoting permutation.

If *pivtng* = 'L': left or row pivoting (matrix must be nonsymmetric).

If *pivtng* = 'R': right or column pivoting (matrix must be nonsymmetric).

If *pivtng* = 'B' or 'F': both or full pivoting, i.e., on both sides. In this case, *m* must equal *n*.

If two calls to ?latmr both have full bandwidth (*kl* = *m* – 1 and *ku* = *n*–1), and differ only in the *pivtng* and *pack* parameters, then the matrices generated differs only in the order of the rows and columns, and otherwise contain the same data. This consistency cannot be maintained with less than full bandwidth.

ipivot

INTEGER. Array, size (*n* or *m*) This array specifies the permutation used. After the basic matrix is generated, the rows, columns, or both are permuted.

If row pivoting is selected, ?latmr starts with the last row and interchanges row *m* and row *ipivot*(*m*), then moves to the next-to-last row, interchanging rows (*m*–1) and row *ipivot*(*m*–1), and so on. In terms of "2-cycles", the permutation is (1 *ipivot*(1)) (2 *ipivot*(2)) ... (*mipivot*(*m*)) where the rightmost cycle is applied first. This is the inverse of the effect of pivoting in LINPACK. The idea is that factoring (with pivoting) an identity matrix which has been inverse-pivoted in this way should result in a pivot vector identical to *ipivot*. Not referenced if *pivtng* = 'N'.

sparse

REAL for slatmr,

DOUBLE PRECISION for dlatmr,

REAL for clatmr,

DOUBLE PRECISION for zlatmr,

On entry, specifies the sparsity of the matrix if a sparse matrix is to be generated. *sparse* should lie between 0 and 1. To generate a sparse matrix, for each matrix entry a uniform (0, 1) random number *x* is generated and compared to sparse; if *x* is larger the matrix entry is unchanged and if *x* is smaller the entry is set to zero. Thus on the average a fraction *sparse* of the entries is set to zero.

kl

INTEGER. On entry, specifies the lower bandwidth of the matrix. For example, *kl* = 0 implies upper triangular, *kl* = 1 implies upper Hessenberg, and *kl* at least *m*-1 implies the matrix is not banded. Must equal *ku* if matrix is symmetric or Hermitian.

ku

INTEGER. On entry, specifies the upper bandwidth of the matrix. For example, *ku* = 0 implies lower triangular, *ku* = 1 implies lower Hessenberg, and *ku*at least *n*-1 implies the matrix is not banded. Must equal *kl* if matrix is symmetric or Hermitian.

anorm

REAL for slatmr,

DOUBLE PRECISION for dlatmr,

REAL for clatmr,

DOUBLE PRECISION for zlatmr,

On entry, specifies maximum entry of output matrix (output matrix is multiplied by a constant so that its largest absolute entry equal *anorm*) if *anorm* is nonnegative. If *anorm* is negative no scaling is done.

pack

for slatmr,

for dlatmr,

for `clatmr`,

for `zlatmr`,

On entry, specifies packing of matrix as follows:

If *pack* = `'N'`: no packing

If *pack* = `'U'`: zero out all subdiagonal entries (if symmetric or Hermitian)

If *pack* = `'L'`: zero out all superdiagonal entries (if symmetric or Hermitian)

If *pack* = `'C'`: store the upper triangle columnwise (only if matrix symmetric or Hermitian or square upper triangular)

If *pack* = `'R'`: store the lower triangle columnwise (only if matrix symmetric or Hermitian or square lower triangular) (same as upper half rowwise if symmetric) (same as conjugate upper half rowwise if Hermitian)

If *pack* = `'B'`: store the lower triangle in band storage scheme (only if matrix symmetric or Hermitian)

If *pack* = `'Q'`: store the upper triangle in band storage scheme (only if matrix symmetric or Hermitian)

If *pack* = `'Z'`: store the entire matrix in band storage scheme (pivoting can be provided for by using this option to store *A* in the trailing rows of the allocated storage)

Using these options, the various LAPACK packed and banded storage schemes can be obtained:

LAPACK storage scheme	Value of *pack*
GB	`'Z'`
PB, HB or TB	`'B'` or `'Q'`
PP, HP or TP	`'C'` or `'R'`

If two calls to `?latmr` differ only in the pack parameter, they generate mathematically equivalent matrices.

lda INTEGER. On entry, *lda* specifies the first dimension of *a* as declared in the calling program.

If *pack* = `'N', 'U' or 'L'`, *lda* must be at least *max*`( 1, m )`.

If *pack* = `'C' or 'R'`, *lda* must be at least 1.

If *pack* = `'B', or 'Q'`, *lda* must be `min(` *ku* `+ 1,` *n* `)`.

If *pack* = `'Z'`, *lda* must be at least *kuu* `+` *kll* `+ 1`, where *kuu* = `min(` *ku*`,` *n*`-1 )` and *kll* = `min(` *kl*`,` *n*`-1 )`.

iwork INTEGER. Array, size (*n* or *m*). Workspace. Not referenced if *pivtng* = `'N'`. Changed on exit.

Output Parameters

iseed On exit, the seed is changed.

d May be changed on exit if *mode* is nonzero.

dl On exit, array is changed.

dr On exit, array is changed.

a `REAL` for `slatmr`,

`DOUBLE PRECISION` for `dlatmr`,

`COMPLEX` for `clatmr`,

`DOUBLE COMPLEX` for `zlatmr`,

On exit, *a* is the desired test matrix. Only those entries of *a* which are significant on output is referenced (even if *a* is in packed or band storage format). The unoccupied corners of *a* in band format are zeroed out.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -1, *m* is negative or unequal to *n* and *sym* = 'S' or 'H'.

If *info* = -2, *n* is negative .

If *info* = -3, *dist* is an illegal string.

If *info* = -5, *sym* is an illegal string..

If *info* = -7, *mode* is not in range -6 to 6.

If *info* = -8, *cond* is less than 1.0, and *mode* is neither -6, 0 nor 6.

If *info* = -10, *mode* is neither -6, 0 nor 6 and *rsign* is an illegal string.

If *info* = -11, *grade* is an illegal string, or *grade* = 'E' and *m* is not equal to *n*, or *grade*='L', 'R', 'B', 'S' or 'E' and *sym* = 'H', or *grade* = 'L', 'R', 'B', 'H' or 'E' and *sym* = 'S'

If *info* = -12,*grade* = 'E'and *dl* contains zero .

If *info* = -13, *model* is not in range -6 to 6 and *grade* = 'L', 'B', 'H', 'S' or 'E' .

If *info* = -14, *condl* is less than 1.0, *grade* = 'L', 'B', 'H', 'S' or 'E', and *model* is neither -6, 0 nor 6.

If *info* = -16, *moder* is not in range -6 to 6 and *grade* = 'R' or 'B' .

If *info* = -17, *condr* is less than 1.0, *grade* = 'R' or 'B', and *moder* is neither -6, 0 nor 6 .

If *info* = -18, *pivtng* is an illegal string, or *pivtng* = 'B' or 'F' and *m* is not equal to *n*, or *pivtng* = 'L' or 'R' and *sym* = 'S' or 'H'.

If *info* = -19, *ipivot* contains out of range number and *pivtng* is not equal to 'N' .

If *info* = -20, *kl* is negative.

If *info* = -21, *ku* is negative, or *sym* = 'S' or 'H' and *ku* not equal to *kl* .

If *info* = -22, *sparse* is not in range 0 to 1.

If *info* = -24, *pack* is an illegal string, or *pack* = 'U', 'L', 'B' or 'Q' and *sym* = 'N', or *pack* = 'C' and *sym* = 'N' and either *kl* is not equal to 0 or *n* is not equal to *m*, or *pack* = 'R' and *sym* = 'N', and either *ku* is not equal to 0 or *n* is not equal to *m* .

If *info* = -26, *lda* is too small .

If *info* = 1, error return from ?latm1 (computing *D*) .

If *info* = 2, cannot scale to *dmax* (max. entry is 0) .

If *info* = 3, error return from ?latm1(computing *dl*) .

If *info* = 4, error return from ?latm1(computing *dr*) .

If *info* = 5, *anorm* is positive, but matrix constructed prior to attempting to scale it to have norm *anorm*, is zero .

?latdf

Uses the LU factorization of the n-by-n matrix computed by ?getc2 *and computes a contribution to the reciprocal Dif-estimate.*

Syntax

```
call slatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call dlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call clatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call zlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
```

Include Files

- mkl.fi

Description

The routine ?latdf uses the *LU* factorization of the *n*-by-*n* matrix *Z* computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate by solving *Z*x = b* for *x*, and choosing the right-hand side *b* such that the norm of *x* is as large as possible. On entry *rhs = b* holds the contribution from earlier solved sub-systems, and on return *rhs = x*.

The factorization of *Z* returned by ?getc2 has the form *Z = P*L*U*Q*, where *P* and *Q* are permutation matrices. *L* is lower triangular with unit diagonal elements and *U* is upper triangular.

Input Parameters

ijob

INTEGER.

ijob = 2: First compute an approximative null-vector *e* of *Z* using ?gecon, *e* is normalized, and solve for *Z*x = ±e-f*with the sign giving the greater value of 2-norm(*x*). This option is about 5 times as expensive as default.

ijob≠ 2 (default): Local look ahead strategy where all entries of the right-hand side *b* is chosen as either +1 or -1 .

n INTEGER. The number of columns of the matrix *Z*.

z REAL for slatdf/clatdf

DOUBLE PRECISION for dlatdf/zlatdf.

Array, DIMENSION (*ldz*, *n*)

On entry, the *LU* part of the factorization of the *n*-by-*n* matrix *Z* computed by ?getc2: *Z = P*L*U*Q*.

ldz INTEGER. The leading dimension of the array *Z*. *lda*≥ max(1, *n*).

rhs REAL for slatdf/clatdf

DOUBLE PRECISION for dlatdf/zlatdf.

Array, DIMENSION (*n*).

On entry, *rhs* contains contributions from other subsystems.

rdsum REAL for slatdf/clatdf

DOUBLE PRECISION for dlatdf/zlatdf.

On entry, the sum of squares of computed contributions to the *D*if-estimate under computation by ?tgsyL, where the scaling factor *rdscal* has been factored out. If *trans* = 'T', *rdsum* is not touched.

Note that *rdsum* only makes sense when ?tgsy2 is called by ?tgsyL.

rdscal REAL for slatdf/clatdf

DOUBLE PRECISION for dlatdf/zlatdf.

On entry, scaling factor used to prevent overflow in *rdsum*.

If *trans* = *T*', *rdscal* is not touched.

Note that *rdscal* only makes sense when ?tgsy2 is called by ?tgsyL.

ipiv INTEGER.

Array, DIMENSION (*n*).

The pivot indices; for 1 ≤ i ≤ *n*, row *i* of the matrix has been interchanged with row *ipiv*(*i*).

jpiv INTEGER.

Array, DIMENSION (*n*).

The pivot indices; for 1 ≤j≤ *n*, column *j* of the matrix has been interchanged with column *jpiv*(*j*).

Output Parameters

rhs On exit, *rhs* contains the solution of the subsystem with entries according to the value of *ijob*.

rdsum On exit, the corresponding sum of squares updated with the contributions from the current sub-system.

If *trans* = 'T', *rdsum* is not touched.

rdscal — On exit, *rdscal* is updated with respect to the current contributions in *rdsum*.

If `trans = 'T'`, *rdscal* is not touched.

?latps

Solves a triangular system of equations with the matrix held in packed storage.

Syntax

```
call slatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call dlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call clatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call zlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
```

Include Files

- `mkl.fi`

Description

The routine `?latps` solves one of the triangular systems

$A*x = s*b$, or $A^T*x = s*b$, or $A^H*x = s*b$ (for complex flavors)

with scaling to prevent overflow, where *A* is an upper or lower triangular matrix stored in packed form. Here A^T denotes the transpose of *A*, A^H denotes the conjugate transpose of *A*, *x* and *b* are *n*-element vectors, and *s* is a scaling factor, usually less than or equal to 1, chosen so that the components of *x* will be less than the overflow threshold. If the unscaled problem does not cause overflow, the Level 2 BLAS routine `?tpsv` is called. If the matrix *A* is singular ($A(j, j) = 0$ for some *j*), then s is set to 0 and a non-trivial solution to $A*x = 0$ is returned.

Input Parameters

uplo — `CHARACTER*1`.

Specifies whether the matrix *A* is upper or lower triangular.

= `'U'`: upper triangular

= `'L'`: uower triangular

trans — `CHARACTER*1`.

Specifies the operation applied to *A*.

= `'N'`: solve $A*x = s*b$ (no transpose)

= `'T'`: solve $A^T*x = s*b$ (transpose)

= `'C'`: solve $A^H*x = s*b$ (conjugate transpose)

diag — `CHARACTER*1`.

Specifies whether the matrix *A* is unit triangular.

= *'N'*: non-unit triangular

= `'U'`: unit triangular

normin CHARACTER*1.

Specifies whether *cnorm* is set.

= 'Y': *cnorm* contains the column norms on entry;

= 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in cnorm.

n INTEGER. The order of the matrix *A*. $n \geq 0$.

ap REAL for slatps

DOUBLE PRECISION for dlatps

COMPLEX for clatps

DOUBLE COMPLEX for zlatps.

Array, DIMENSION ($n(n+1)/2$).

The upper or lower triangular matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;

if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.

x REAL for slatpsDOUBLE PRECISION for dlatps

COMPLEX for clatps

DOUBLE COMPLEX for zlatps.

Array, DIMENSION (*n*)

On entry, the right hand side *b* of the triangular system.

cnorm REAL for slatps/clatps

DOUBLE PRECISION for dlatps/zlatps.

Array, DIMENSION (*n*).

If *normin* = 'Y', *cnorm* is an input argument and *cnorm*(*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*.

If *trans* = 'N', *cnorm*(*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

Output Parameters

x On exit, *x* is overwritten by the solution vector *x*.

scale REAL for slatps/clatps

DOUBLE PRECISION for dlatps/zlatps.

The scaling factor *s* for the triangular system as described above.

If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to *A*x* = 0.

cnorm If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

info	INTEGER. = 0: successful exit < 0: if *info* = -*k*, the *k*-th argument had an illegal value

?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
call slatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call dlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call clatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call zlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
```

Include Files

- mkl.fi

Description

The routine ?latrd reduces *nb* rows and columns of a real symmetric or complex Hermitian matrix *A* to symmetric/Hermitian tridiagonal form by an orthogonal/unitary similarity transformation Q^T*A*Q for real flavors, Q^H*A*Q for complex flavors, and returns the matrices *V* and *W* which are needed to apply the transformation to the unreduced part of *A*.

If *uplo* = 'U', ?latrd reduces the last *nb* rows and columns of a matrix, of which the upper triangle is supplied;

if *uplo* = 'L', ?latrd reduces the first *nb* rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by ?sytrd/?hetrd.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix *A* is stored: = 'U': upper triangular = 'L': lower triangular
n	INTEGER. The order of the matrix *A*.
nb	INTEGER. The number of rows and columns to be reduced.
a	REAL for slatrd DOUBLE PRECISION for dlatrd COMPLEX for clatrd DOUBLE COMPLEX for zlatrd.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric/Hermitian matrix *A*

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*. *lda* ≥ (1,*n*).

ldw INTEGER.

The leading dimension of the output array *w*. *ldw* ≥ max(1,*n*).

Output Parameters

a On exit, if *uplo* = 'U', the last *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of *a*; the elements above the diagonal with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;

if *uplo* = 'L', the first *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of *a*; the elements below the diagonal with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

e REAL for slatrd/clatrd

DOUBLE PRECISION for dlatrd/zlatrd.

If *uplo* = 'U', *e*(*n*-*nb*:*n*-1) contains the superdiagonal elements of the last *nb* columns of the reduced matrix;

if *uplo* = 'L', *e*(1:*nb*) contains the subdiagonal elements of the first *nb* columns of the reduced matrix.

tau REAL for slatrd

DOUBLE PRECISION for dlatrd

COMPLEX for clatrd

DOUBLE COMPLEX for zlatrd.

Array, DIMENSION (*lda*, *n*).

The scalar factors of the elementary reflectors, stored in *tau*(*n*-*nb*:*n*-1) if *uplo* = 'U', and in *tau*(1:*nb*) if *uplo* = 'L'.

w REAL for slatrd

DOUBLE PRECISION for dlatrd

COMPLEX for clatrd

DOUBLE COMPLEX for zlatrd.

Array, DIMENSION (*ldw*, *n*).

The *n*-by-*nb* matrix *W* required to update the unreduced part of *A*.

Application Notes

If `uplo = 'U'`, the matrix *Q* is represented as a product of elementary reflectors

`Q = H(n)*H(n-1)*...*H(n-nb+1)`

Each *H*(*i*) has the form

`H(i) = I - tau*v*v'`

where *tau* is a real/complex scalar, and *v* is a real/complex vector with `v(i:n) = 0` and `v(i-1) = 1`; `v(1: i-1)` is stored on exit in `a(1: i-1, i)`, and *tau* in `tau(i-1)`.

If `uplo = 'L'`, the matrix *Q* is represented as a product of elementary reflectors

`Q = H(1)*H(2)*...*H(nb)`

Each *H*(*i*) has the form `H(i) = I - tau*v*v'`

where *tau* is a real/complex scalar, and *v* is a real/complex vector with `v(1: i) = 0` and `v(i+1) = 1`; `v( i +1:n)` is stored on exit in `a(i+1:n, i)`, and *tau* in *tau*(*i*).

The elements of the vectors *v* together form the *n*-by-*nb* matrix *V* which is needed, with *W*, to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2k update of the form:

`A := A - VW' - WV'`.

The contents of *a* on exit are illustrated by the following examples with `n = 5` and `nb = 2`:

if $uplo = $'U':

$$\begin{bmatrix} a & a & a & v_4 & v_4 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & & d & 1 \\ & & & & d \end{bmatrix}$$

if $uplo = $'L'

$$\begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where *d* denotes a diagonal element of the reduced matrix, *a* denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining *H*(*i*).

?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call slatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call dlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
```

```
call clatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call zlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
```

Include Files

- mkl.fi

Description

The routine solves one of the triangular systems

$A*x = s*b$, or $A^T*x = s*b$, or $A^H*x = s*b$ (for complex flavors)

with scaling to prevent overflow. Here A is an upper or lower triangular matrix, A^T denotes the transpose of A, A^H denotes the conjugate transpose of A, x and b are n-element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine ?trsv is called. If the matrix A is singular ($A(j,j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $A*x = 0$ is returned.

Input Parameters

uplo

CHARACTER*1.

Specifies whether the matrix A is upper or lower triangular.

= 'U': Upper triangular

= 'L': Lower triangular

trans

CHARACTER*1.

Specifies the operation applied to A.

= 'N': solve $A*x = s*b$ (no transpose)

= 'T': solve $A^T*x = s*b$ (transpose)

= 'C': solve $A^H*x = s*b$ (conjugate transpose)

diag

CHARACTER*1.

Specifies whether or not the matrix A is unit triangular.

= 'N': non-unit triangular

= 'N': non-unit triangular

normin

CHARACTER*1.

Specifies whether *cnorm* has been set or not.

= 'Y': *cnorm* contains the column norms on entry;

= 'N': *cnorm* is not set on entry. O

n exit, the norms will be computed and stored in *cnorm*.

n

INTEGER. The order of the matrix A. $n \geq 0$

a

REAL for slatrs

DOUBLE PRECISION for dlatrs

COMPLEX for clatrs

DOUBLE COMPLEX for zlatrs.

Array, DIMENSION (*lda*, *n*). Contains the triangular matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *A* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *A* is not referenced.

If *diag* = 'U', the diagonal elements of *A* are also not referenced and are assumed to be 1.

lda INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1, *n*).

x REAL for slatrs

DOUBLE PRECISION for dlatrs

COMPLEX for clatrs

DOUBLE COMPLEX for zlatrs.

Array, DIMENSION (*n*).

On entry, the right hand side *b* of the triangular system.

cnorm REAL for slatrs/clatrs

DOUBLE PRECISION for dlatrs/zlatrs.

Array, DIMENSION (*n*).

If *normin* = 'Y', *cnorm* is an input argument and *cnorm* (*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*.

If *trans* = 'N', *cnorm* (*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

Output Parameters

x On exit, *x* is overwritten by the solution vector *x*.

scale REAL for slatrs/clatrs

DOUBLE PRECISION for dlatrs/zlatrs.

Array, DIMENSION (*lda*, *n*). The scaling factor *s* for the triangular system as described above.

If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to *A***x* = 0.

cnorm If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

info INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument had an illegal value

Application Notes

A rough bound on *x* is computed; if that is less than overflow, `?trsv` is called, otherwise, specific code is used which checks for possible overflow or divide-by-zero at every operation.

A columnwise scheme is used for solving $Ax = b$. The basic algorithm if *A* is lower triangular is

```
x[1:n] := b[1:n]
for j = 1, ..., n
x(j) := x(j) / A(j,j)
x[j+1:n] := x[j+1:n] - x(j)*a[j+1:n,j]
end
```

Define bounds on the components of *x* after *j* iterations of the loop:

$M(j)$ = bound on $x[1:j]$

$G(j)$ = bound on $x[j+1:n]$

Initially, let $M(0) = 0$ and $G(0) = \max\{x(i), i=1,...,n\}$.

Then for iteration $j+1$ we have

$M(j+1) \le G(j) / |a(j+1,j+1)|$

$G(j+1) \le G(j) + M(j+1)*|a[j+2:n,j+1]|$

$\le G(j)(1 + cnorm(j+1)/|a(j+1,j+1)|,$

where *cnorm*(*j*+1) is greater than or equal to the infinity-norm of column *j*+1 of *a*, not counting the diagonal. Hence

$$G(j) \le G(0) \prod_{1 \le i \le j} (1 + cnorm(i)/|A(i,i)|)$$

and

$$|x(j)| \le (G(0)/|A(j,j)|) \prod_{1 \le i \le j} (1 + cnorm(i)/|A(i,i)|)$$

Since $|x(j)| \le M(j)$, we use the Level 2 BLAS routine `?trsv` if the reciprocal of the largest $M(j)$, $j=1,..,n$, is larger than $\max(underflow, 1/overflow)$.

The bound on *x*(*j*) is also used to determine when a step in the columnwise method can be performed without fear of overflow. If the computed bound is greater than a large constant, *x* is scaled to prevent overflow, but if the bound overflows, *x* is set to 0, *x*(*j*) to 1, and scale to 0, and a non-trivial solution to $Ax = 0$ is found.

Similarly, a row-wise scheme is used to solve $A^Tx = b$ or $A^Hx = b$. The basic algorithm for *A* upper triangular is

```
for j = 1, ..., n
x(j) := ( b(j) - A[1:j-1,j]' x[1:j-1]) / A(j,j)
end
```

We simultaneously compute two bounds

$G(j) = \text{bound on } (b(i) - A[1:i-1,i]'*x[1:i-1]),\ 1 \le i \le j$

$M(j) = \text{bound on } x(i),\ 1 \le i \le j$

The initial values are $G(0) = 0$, $M(0) = \max\{ b(i),\ i=1,..,n\}$, and we add the constraint $G(j) \ge G(j-1)$ and $M(j) \ge M(j-1)$ for $j \ge 1$.

Then the bound on $x(j)$ is

$M(j) \le M(j-1) *(1 + cnorm(j)) / | A(j,j)|$

$$\le M(0) \prod_{1 \le i \le j} (1 + cnorm(i)/|A(i,i)|)$$

and we can safely call ?trsv if $1/M(n)$ and $1/G(n)$ are both greater than $\max(underflow,\ 1/overflow)$.

?latrz

Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.

Syntax

```
call slatrz( m, n, l, a, lda, tau, work )
call dlatrz( m, n, l, a, lda, tau, work )
call clatrz( m, n, l, a, lda, tau, work )
call zlatrz( m, n, l, a, lda, tau, work )
```

Include Files

- mkl.fi

Description

The routine ?latrz factors the *m*-by-(*m*+*l*) real/complex upper trapezoidal matrix

$[A1\ A2] = [A(1:m,1:m)\ A(1:\ m,\ n-l+1:n)]$

as (*R* 0)* *Z*, by means of orthogonal/unitary transformations. *Z* is an (*m*+*l*)-by-(*m*+*l*) orthogonal/unitary matrix and *R* and *A*1 are *m*-by -*m* upper triangular matrices.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. $m \ge 0$.
n	INTEGER. The number of columns of the matrix *A*. $n \ge 0$.
l	INTEGER. The number of columns of the matrix *A* containing the meaningful part of the Householder vectors. $n-m \ge l \ge 0$.
a	REAL for slatrz DOUBLE PRECISION for dlatrz COMPLEX for clatrz DOUBLE COMPLEX for zlatrz.

Array, DIMENSION (*lda*, *n*).

On entry, the leading *m*-by-*n* upper trapezoidal part of the array *a* must contain the matrix to be factorized.

lda INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*m*).

work REAL for slatrz

DOUBLE PRECISION for dlatrz

COMPLEX for clatrz

DOUBLE COMPLEX for zlatrz.

Workspace array, DIMENSION (*m*).

Output Parameters

a On exit, the leading *m*-by-*m* upper triangular part of *a* contains the upper triangular matrix *R*, and elements *n*-*l*+1 to *n* of the first *m* rows of *a*, with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

tau REAL for slatrz

DOUBLE PRECISION for dlatrz

COMPLEX for clatrz

DOUBLE COMPLEX for zlatrz.

Array, DIMENSION (*m*).

The scalar factors of the elementary reflectors.

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *z*(*k*), which is used to introduce zeros into the (*m* - *k* + 1)-th row of *A*, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where for real flavors

$$T(k) = I - tau^* u(k)^* T(k)^T, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

and for complex flavors

$$T(k) = I - tau^* u(k)^* T(k)^H, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and *z*(*k*) is an *l*-element vector. *tau* and *z*(*k*) are chosen to annihilate the elements of the *k*-th row of *A*2.

The scalar *tau* is returned in the *k*-th element of *tau* and the vector *u*(*k*) in the *k*-th row of *A*2, such that the elements of *z*(*k*) are in `a(k, l+1), ..., a(k, n)`.

The elements of *r* are returned in the upper triangular part of *A*1.

Z is given by

`Z = Z(1)*Z(2)*...*Z(m)`.

?latsqr

Computes a blocked Tall-Skinny QR matrix factorization.

```
call slatsqr(m, n, mb, nb, a, lda, t, ldt, work, lwork, info)
call dlatsqr(m, n, mb, nb, a, lda, t, ldt, work, lwork, info)
call clatsqr(m, n, mb, nb, a, lda, t, ldt, work, lwork, info)
call zlatsqr(m, n, mb, nb, a, lda, t, ldt, work, lwork, info)
```

Description

`?latsqr` computes a blocked Tall-Skinny QR (TSQR) factorization of an `m`-by-`n` matrix *A*, where `m≥n`: *A* = *Q***R*.

TSQR performs QR by a sequence of orthogonal transformations, representing *Q* as a product of other orthogonal matrices

Q = *Q*(1) * *Q*(2) * . . . * *Q*(*k*)

where each *Q*(*i*) zeros out subdiagonal entries of a block of `mb` rows of *A*:

Q(1) zeros out the subdiagonal entries of rows 1:MB of *A*,

$Q(2)$ zeros out the bottom mb - n rows of rows [1:n, mb + 1:2*mb - n] of A,

$Q(3)$ zeros out the bottom mb - n rows of rows [1:n, 2*mb - n + 1:3*mb - 2*n] of A . . . .

$Q(1)$ is computed by `geqrt`, which represents $Q(1)$ by Householder vectors stored under the diagonal of rows 1:mb of a, and by upper triangular block reflectors, stored in array t(1:ldt, 1:n). For more information see `geqrt`.

$Q(i)$ for $i > 1$ is computed by `tpqrt`, which represents $Q(i)$ by Householder vectors stored in rows [(i - 1)*(mb - n) + n + 1:i*(mb - n) + n] of a, and by upper triangular block reflectors, stored in array t(1:ldt, (i - 1)*n + 1:i*n). The last $Q(k)$ may use fewer rows. For more information, see `tpqrt`. For more details of the overall algorithm, see [DEMMEL12]

Input Parameters

m	`INTEGER`. The number of rows of the matrix A. $m \geq 0$.
n	`INTEGER`. The number of columns of the matrix A. $m \geq n \geq 0$.
mb	`INTEGER`. The row block size to be used in the blocked QR. $mb > n$.
nb	`INTEGER`. The column block size to be used in the blocked QR. $n \geq nb \geq 1$.

a	`REAL` for `slatsqr`
	`DOUBLE PRECISION` for `dlatsqr`
	`COMPLEX` for `clatsqr`
	`COMPLEX*16` for `zlatsqr`
	Array of size (lda, n). On entry, the m-by-n matrix A.

lda	`INTEGER`. The leading dimension of the array a. $lda \geq$ max(1, m).
ldt	`INTEGER`. The leading dimension of the array t. $ldt \geq nb$.
$lwork$	`INTEGER`. The dimension of the array $work$. $lwork \geq nb*n$. If $lwork$ = -1, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

Output Parameters

a	On exit, the elements on and above the diagonal of the array contain the n-by-n upper triangular matrix R and the elements below the diagonal represent Q by the columns of blocked V.

t	`REAL` for `slatsqr`
	`DOUBLE PRECISION` for `dlatsqr`
	`COMPLEX` for `clatsqr`
	`COMPLEX*16` for `zlatsqr`
	Array of size (ldt, n * *Number_of_row_blocks*) where *Number_of_row_blocks* = ceiling((m - n)/(mb - n)) The blocked upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks.

work REAL for slatsqr

DOUBLE PRECISION for dlatsqr

COMPLEX for clatsqr

COMPLEX*16 for zlatsqr

Workspace array of size (max(1, *lwork*)).

info INTEGER.

info = 0: successful exit.

info < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?lauu2

Computes the product $U*U^T$ ($U*U^H$) or L^T*L (L^H*L), where U and L are upper or lower triangular matrices (unblocked algorithm).

Syntax

```
call slauu2( uplo, n, a, lda, info )
call dlauu2( uplo, n, a, lda, info )
call clauu2( uplo, n, a, lda, info )
call zlauu2( uplo, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine ?lauu2 computes the product $U*U^T$ or L^T*L for real flavors, and $U*U^H$ or L^H*L for complex flavors. Here the triangular factor *U* or *L* is stored in the upper or lower triangular part of the array *a*.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in *A*.

If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in *A*.

This is the unblocked form of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo CHARACTER*1.

Specifies whether the triangular factor stored in the array *a* is upper or lower triangular:

= 'U': Upper triangular

= 'L': Lower triangular

n INTEGER. The order of the triangular factor *U* or *L*. $n \geq 0$.

a REAL for slauu2

DOUBLE PRECISION for dlauu2

COMPLEX for clauu2

DOUBLE COMPLEX for zlauu2.

Array, DIMENSION (*lda*, *n*). On entry, the triangular factor *U* or *L*.

lda INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1, *n*).

Output Parameters

a On exit,

if *uplo* = 'U', then the upper triangle of *a* is overwritten with the upper triangle of the product $U*U^T$ ($U*U^H$);

if *uplo* = 'L', then the lower triangle of *a* is overwritten with the lower triangle of the product L^T*L (L^H*L).

info INTEGER.

= 0: successful exit

< 0: if *info* = −*k*, the *k*-th argument had an illegal value

?lauum

Computes the product $U*U^T$ ($U*U^H$) or L^T*L (L^H*L), where U and L are upper or lower triangular matrices (blocked algorithm).

Syntax

```
call slauum( uplo, n, a, lda, info )
call dlauum( uplo, n, a, lda, info )
call clauum( uplo, n, a, lda, info )
call zlauum( uplo, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine ?lauum computes the product $U*U^T$ or L^T*L for real flavors, and $U*U^H$ or L^H*L for complex flavors. Here the triangular factor *U* or *L* is stored in the upper or lower triangular part of the array *a*.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in *A*.

If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in *A*.

This is the blocked form of the algorithm, calling BLAS Level 3 Routines.

Input Parameters

The data types are given for the Fortran interface.

uplo CHARACTER*1.

Specifies whether the triangular factor stored in the array *a* is upper or lower triangular:

= 'U': Upper triangular

= 'L': Lower triangular

n — INTEGER. The order of the triangular factor *U* or *L*. $n \geq 0$.

a — REAL for slauum

DOUBLE PRECISION for dlauum

COMPLEX for clauum

DOUBLE COMPLEX for zlauum .

Array of size (*lda*, *n*).

On entry, the triangular factor *U* or *L*.

lda — INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a

On exit,

if *uplo* = 'U', then the upper triangle of *a* is overwritten with the upper triangle of the product $U * U^T$($U * U^H$);

if *uplo* = 'L', then the lower triangle of *a* is overwritten with the lower triangle of the product $L^T * L$ ($L^H * L$).

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = *-k*, the *k*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?orbdb1/?unbdb1

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

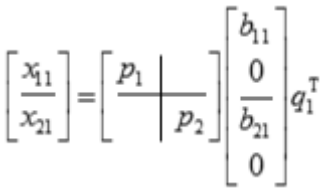
```
call sorbdb1( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
call dorbdb1( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
call cunbdb1( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
call zunbdb1( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines ?orbdb1/?unbdb1 simultaneously bidiagonalize the blocks of a tall and skinny matrix *X* with orthonormal columns:



The size of x_{11} is p by q, and x_{12} is $(m - p)$ by q. q must not be larger than p, m-p, or m-q.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$	?orbdb1/?unbdb1
$p \leq \min(q, m - p, m - q)$	?orbdb2/?unbdb2
$m - p \leq \min(p, q, m - q)$	?orbdb3/?unbdb3
$m - q \leq \min(p, q, m - p)$	?orbdb4/?unbdb4

The orthogonal/unitary matrices p_1, p_2, and q_1 are p-by-p, $(m-p)$-by-$(m-p)$, $(m-q)$-by-$(m-q)$, respectively.

p_1, p_2, and q_1 are represented as products of elementary reflectors. See the description of ?orcsd2by1/?uncsd2by1 for details on generating p_1, p_2, and q_1 using ?orgqr and ?orglq.

The upper-bidiagonal matrices b_{11} and b_{12} of size q by q are represented implicitly by angles *theta*(1), ..., *theta*(q) and *phi*(1), ..., *phi*(q-1). Every entry in each bidiagonal band is a product of a sine or cosine of *theta* with a sine or cosine of *phi*. See [Sutton09] or the description of ?orcsd/?uncsd for details.

Input Parameters

m
INTEGER. The number of rows in x_{11} plus the number of rows in x_{21}.

p
INTEGER. The number of rows in x_{11}. $0 \leq p \leq m$.

q
INTEGER. The number of columns in x_{11} and x_{21}. $0 \leq q \leq \min(p, m-p, m-q)$.

x11
REAL for sorbdb1
DOUBLE PRECISION for dorbdb1
COMPLEX for cunbdb1
DOUBLE COMPLEX for zunbdb1
Array, DIMENSION (*ldx11*,*q*).
On entry, the top block of the orthogonal/unitary matrix to be reduced.

ldx11
INTEGER. The leading dimension of the array X_{11}. $ldx11 \geq p$.

x21
REAL for sorbdb1
DOUBLE PRECISION for dorbdb1
COMPLEX for cunbdb1
DOUBLE COMPLEX for zunbdb1
Array, DIMENSION (*ldx21*,*q*).
On entry, the bottom block of the orthogonal/unitary matrix to be reduced.

ldx21
INTEGER. The leading dimension of the array X_{21}. $ldx21 \geq m-p$.

work

REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

COMPLEX for cunbdb1

DOUBLE COMPLEX for zunbdb1

Workspace array, DIMENSION (*lwork*).

lwork

INTEGER. The size of the *work* array. *lwork*≥*m-q*

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x11

The columns of tril(*x11*) specify reflectors for p_1 and the rows of triu(*x11*,1) specify reflectors for q_1, where tril(*A*) denotes the lower triangle of *A*, and triu(*A*) denotes the upper triangle of *A*.

x21

On exit, the columns of tril(*x21*) specify the reflectors for p_2

theta

REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

COMPLEX for cunbdb1

DOUBLE COMPLEX for zunbdb1

Array, DIMENSION (*q*). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi

REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

COMPLEX for cunbdb1

DOUBLE COMPLEX for zunbdb1

Array, DIMENSION (*q*-1). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1

REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

COMPLEX for cunbdb1

DOUBLE COMPLEX for zunbdb1

Array, DIMENSION (*p*).

Scalar factors of the elementary reflectors that define p_1.

taup2

REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

	COMPLEX for cunbdb1 DOUBLE COMPLEX for zunbdb1 Array, DIMENSION (*m*-*p*). Scalar factors of the elementary reflectors that define p_2.
tauq1	REAL for sorbdb1 DOUBLE PRECISION for dorbdb1 COMPLEX for cunbdb1 DOUBLE COMPLEX for zunbdb1 Array, DIMENSION (*q*). Scalar factors of the elementary reflectors that define q_1.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value.

See Also

?orcsd/?uncsd Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

?orcsd2by1/?uncsd2by1 Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

?orbdb2/?unbdb2 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb3/?unbdb3 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb4/?unbdb4 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb5/?unbdb5 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

?orbdb6/?unbdb6 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

xerbla

?orbdb2/?unbdb2

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

```
call sorbdb2( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
call dorbdb2( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
call cunbdb2( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
call zunbdb2( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines ?orbdb2/?unbdb2 simultaneously bidiagonalize the blocks of a tall and skinny matrix *X* with orthonormal columns:

$$\begin{bmatrix} x_{11} \\ \hline x_{21} \end{bmatrix} = \left[\begin{array}{c|c} p_1 & \\ \hline & p_2 \end{array}\right] \begin{bmatrix} b_{11} \\ 0 \\ \hline b_{21} \\ 0 \end{bmatrix} q_1^T$$

The size of x_{11} is p by q, and x_{12} is $(m - p)$ by q. q must not be larger than p, m-p, or m-q.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$	?orbdb1/?unbdb1
$p \leq \min(q, m - p, m - q)$	?orbdb2/?unbdb2
$m - p \leq \min(p, q, m - q)$	?orbdb3/?unbdb3
$m - q \leq \min(p, q, m - p)$	?orbdb4/?unbdb4

The orthogonal/unitary matrices p_1, p_2, and q_1 are p-by-p, $(m-p)$-by-$(m-p)$, $(m-q)$-by-$(m-q)$, respectively.

p_1, p_2, and q_1 are represented as products of elementary reflectors. See the description of ?orcsd2by1/?uncsd2by1 for details on generating p_1, p_2, and q_1 using ?orgqr and ?orglq.

The upper-bidiagonal matrices b_{11} and b_{12} of size p by p are represented implicitly by angles *theta*(1), ..., *theta*(*q*) and *phi*(1), ..., *phi*(*q*-1). Every entry in each bidiagonal band is a product of a sine or cosine of *theta* with a sine or cosine of *phi*. See [Sutton09] or the description of ?orcsd/?uncsd for details.

Input Parameters

m
INTEGER. The number of rows in x_{11} plus the number of rows in x_{21}.

p
INTEGER. The number of rows in x_{11}. $0 \leq p \leq \min(q, m-p, m-q)$.

q
INTEGER. The number of columns in x_{11} and x_{21}. $0 \leq q \leq m$.

x11
REAL for sorbdb2
DOUBLE PRECISION for dorbdb2
COMPLEX for cunbdb2
DOUBLE COMPLEX for zunbdb2
Array, DIMENSION (*ldx11*,*q*).
On entry, the top block of the orthogonal/unitary matrix to be reduced.

ldx11
INTEGER. The leading dimension of the array X_{11}. $ldx11 \geq p$.

x21
REAL for sorbdb2
DOUBLE PRECISION for dorbdb2
COMPLEX for cunbdb2
DOUBLE COMPLEX for zunbdb2
Array, DIMENSION (*ldx21*,*q*).

On entry, the bottom block of the orthogonal/unitary matrix to be reduced.

ldx21 INTEGER. The leading dimension of the array X_{21}. *ldx21*≥*m*-*p*.

work REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Workspace array, DIMENSION (*lwork*).

lwork INTEGER. The size of the *work* array. *lwork*≥*m*-*q*

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x11 On exit: the columns of tril(*x11*) specify reflectors for p_1 and the rows of triu(*x11*,1) specify reflectors for q_1.

x21 On exit, the columns of tril(*x21*) specify the reflectors for p_2

theta REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Array, DIMENSION (*q*). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Array, DIMENSION (*q*-1). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1 REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Array, DIMENSION (*p*).

Scalar factors of the elementary reflectors that define p_1.

taup2 REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Array, DIMENSION (*m*-*p*).

Scalar factors of the elementary reflectors that define p_2.

tauq1

REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Array, DIMENSION (*q*).

Scalar factors of the elementary reflectors that define q_1.

info

INTEGER.

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument has an illegal value.

See Also

?orcsd/?uncsd

?orcsd2by1/?uncsd2by1 Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

?orbdb1/?unbdb1 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb3/?unbdb3 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb4/?unbdb4 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb5/?unbdb5 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

?orbdb6/?unbdb6 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

xerbla

?orbdb3/?unbdb3

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

```
call sorbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
call dorbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
call cunbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
call zunbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines ?orbdb3/?unbdb3 simultaneously bidiagonalize the blocks of a tall and skinny matrix *X* with orthonormal columns:

$$\begin{bmatrix} x_{11} \\ \hline x_{21} \end{bmatrix} = \left[\begin{array}{c|c} p_1 & \\ \hline & p_2 \end{array}\right] \begin{bmatrix} b_{11} \\ 0 \\ \hline b_{21} \\ 0 \end{bmatrix} q_1^T$$

The size of x_{11} is *p* by *q*, and x_{12} is (*m* - *p*) by *q*. *m*-*p* must not be larger than *p*, *q*, or *m*-*q*.

Tall and Skinny Matrix Routines

$q \leq \min(p, m-p, m-q)$	?orbdb1/?unbdb1
$p \leq \min(q, m-p, m-q)$	?orbdb2/?unbdb2
$m-p \leq \min(p, q, m-q)$	?orbdb3/?unbdb3
$m-q \leq \min(p, q, m-p)$	?orbdb4/?unbdb4

The orthogonal/unitary matrices p_1, p_2, and q_1 are *p*-by-*p*, *(m-p)*-by-*(m-p)*, *(m-q)*-by-*(m-q)*, respectively.

p_1, p_2, and q_1 are represented as products of elementary reflectors. See the description of ?orcsd2by1/?uncsd2by1 for details on generating p_1, p_2, and q_1 using ?orgqr and ?orglq.

The upper-bidiagonal matrices b_{11} and b_{12} of size (*m*-*p*) by (*m*-*p*)are represented implicitly by angles *theta*(1), ..., *theta*(*q*) and *phi*(1), ..., *phi*(*q*-1). Every entry in each bidiagonal band is a product of a sine or cosine of *theta* with a sine or cosine of *phi*. See [Sutton09] or the description of ?orcsd/?uncsd for details.

Input Parameters

m
INTEGER. The number of rows in x_{11} plus the number of rows in x_{21}.

p
INTEGER. The number of rows in x_{11}. $0 \leq p \leq m$, $m-p \leq \min(p, q, m-q)$.

q
INTEGER. The number of columns in x_{11} and x_{21}. $0 \leq q \leq m$.

x11
REAL for sorbdb3
DOUBLE PRECISION for dorbdb3
COMPLEX for cunbdb3
DOUBLE COMPLEX for zunbdb3
Array, DIMENSION (*ldx11*,*q*).
On entry, the top block of the orthogonal/unitary matrix to be reduced.

ldx11
INTEGER. The leading dimension of the array X_{11}. $ldx11 \geq p$.

x21
REAL for sorbdb3
DOUBLE PRECISION for dorbdb3

COMPLEX for cunbdb3

DOUBLE COMPLEX for zunbdb3

Array, DIMENSION (*ldx21*,*q*).

On entry, the bottom block of the orthogonal/unitary matrix to be reduced.

ldx21 — INTEGER. The leading dimension of the array X_{21}. *ldx21*≥*m*-*p*.

work — REAL for sorbdb3

DOUBLE PRECISION for dorbdb3

COMPLEX for cunbdb3

DOUBLE COMPLEX for zunbdb3

Workspace array, DIMENSION (*lwork*).

lwork — INTEGER. The size of the *work* array. *lwork*≥*m*-*q*

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x11 — On exit: the columns of tril(*x11*) specify reflectors for p_1 and the rows of triu(*x11*,1) specify reflectors for q_1.

x21 — On exit, the columns of tril(*x21*) specify the reflectors for p_2

theta — REAL for sorbdb3

DOUBLE PRECISION for dorbdb3

COMPLEX for cunbdb3

DOUBLE COMPLEX for zunbdb3

Array, DIMENSION (*q*). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi — REAL for sorbdb3

DOUBLE PRECISION for dorbdb3

COMPLEX for cunbdb3

DOUBLE COMPLEX for zunbdb3

Array, DIMENSION (*q*-1). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1 — REAL for sorbdb3

DOUBLE PRECISION for dorbdb3

COMPLEX for cunbdb3

DOUBLE COMPLEX for zunbdb3

	Array, DIMENSION (p).
	Scalar factors of the elementary reflectors that define p_1.
taup2	REAL for sorbdb3
	DOUBLE PRECISION for dorbdb3
	COMPLEX for cunbdb3
	DOUBLE COMPLEX for zunbdb3
	Array, DIMENSION (m-p).
	Scalar factors of the elementary reflectors that define p_2.
tauq1	REAL for sorbdb3
	DOUBLE PRECISION for dorbdb3
	COMPLEX for cunbdb3
	DOUBLE COMPLEX for zunbdb3
	Array, DIMENSION (q).
	Scalar factors of the elementary reflectors that define q_1.
info	INTEGER.
	= 0: successful exit
	< 0: if *info* = -*i*, the *i*-th argument has an illegal value.

See Also

?orcsd/?uncsd

?orcsd2by1/?uncsd2by1 Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

?orbdb1/?unbdb1 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb2/?unbdb2 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb4/?unbdb4 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb5/?unbdb5 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

?orbdb6/?unbdb6 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

xerbla

?orbdb4/?unbdb4

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

```
call sorbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, phantom, work, lwork, info )
```

```
call dorbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, phantom, work, lwork, info )
```

```
call cunbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, phantom, work, lwork, info )
```

```
call zunbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, phantom,
work, lwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines ?orbdb4/?unbdb4 simultaneously bidiagonalize the blocks of a tall and skinny matrix *X* with orthonormal columns:

$$\begin{bmatrix} x_{11} \\ \hline x_{21} \end{bmatrix} = \left[\begin{array}{c|c} p_1 & \\ \hline & p_2 \end{array}\right] \begin{bmatrix} b_{11} \\ 0 \\ \hline b_{21} \\ 0 \end{bmatrix} q_1^T$$

The size of x_{11} is *p* by *q*, and x_{12} is (*m* - *p*) by *q*. *m*-*q* must not be larger than *q*, *p*, or *m*-*p*.

Tall and Skinny Matrix Routines

$q \leq \min(p, m-p, m-q)$	?orbdb1/?unbdb1
$p \leq \min(q, m-p, m-q)$	?orbdb2/?unbdb2
$m-p \leq \min(p, q, m-q)$	?orbdb3/?unbdb3
$m-q \leq \min(p, q, m-p)$	?orbdb4/?unbdb4

The orthogonal/unitary matrices p_1, p_2, and q_1 are *p*-by-*p*, *(m-p)*-by-*(m-p)*, *(m-q)*-by-*(m-q)*, respectively.

p_1, p_2, and q_1 are represented as products of elementary reflectors. See the description of ?orcsd2by1/?uncsd2by1 for details on generating p_1, p_2, and q_1 using ?orgqr and ?orglq.

The upper-bidiagonal matrices b_{11} and b_{12} of size *(m-q)* by *(m-q)* are represented implicitly by angles *theta*(1), ..., *theta*(*q*) and *phi*(1), ..., *phi*(*q*-1). Every entry in each bidiagonal band is a product of a sine or cosine of *theta* with a sine or cosine of *phi*. See [Sutton09] or the description of ?orcsd/?uncsd for details.

Input Parameters

m
: INTEGER. The number of rows in x_{11} plus the number of rows in x_{21}.

p
: INTEGER. The number of rows in x_{11}. $0 \leq p \leq m$.

q
: INTEGER. The number of columns in x_{11} and x_{21}. $0 \leq q \leq m$ and $0 \leq m-q \leq \min(p, m-p, q)$.

x11
: REAL for sorbdb4
DOUBLE PRECISION for dorbdb4
COMPLEX for cunbdb4
DOUBLE COMPLEX for zunbdb4
Array, DIMENSION (*ldx11*,*q*).
On entry, the top block of the orthogonal/unitary matrix to be reduced.

ldx11
: INTEGER. The leading dimension of the array X_{11}. $ldx11 \geq p$.

x21
REAL for sorbdb4
DOUBLE PRECISION for dorbdb4
COMPLEX for cunbdb4
DOUBLE COMPLEX for zunbdb4
Array, DIMENSION (*ldx21*,*q*).
On entry, the bottom block of the orthogonal/unitary matrix to be reduced.

ldx21
INTEGER. The leading dimension of the array X_{21}. *ldx21*≥*m*-*p*.

work
REAL for sorbdb4
DOUBLE PRECISION for dorbdb4
COMPLEX for cunbdb4
DOUBLE COMPLEX for zunbdb4
Workspace array, DIMENSION (*lwork*).

lwork
INTEGER. The size of the *work* array. *lwork*≥*m*-*q*
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x11
On exit: the columns of tril(*x11*) specify reflectors for p_1 and the rows of triu(*x11*,1) specify reflectors for q_1.

x21
On exit, the columns of tril(*x21*) specify the reflectors for p_2

theta
REAL for sorbdb4
DOUBLE PRECISION for dorbdb4
COMPLEX for cunbdb4
DOUBLE COMPLEX for zunbdb4
Array, DIMENSION (*q*). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi
REAL for sorbdb4
DOUBLE PRECISION for dorbdb4
COMPLEX for cunbdb4
DOUBLE COMPLEX for zunbdb4
Array, DIMENSION (*q*-1). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1
REAL for sorbdb4
DOUBLE PRECISION for dorbdb4

COMPLEX for cunbdb4

DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (*p*).

Scalar factors of the elementary reflectors that define p_1.

taup2

REAL for sorbdb4

DOUBLE PRECISION for dorbdb4

COMPLEX for cunbdb4

DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (*m*-*p*).

Scalar factors of the elementary reflectors that define p_2.

tauq1

REAL for sorbdb4

DOUBLE PRECISION for dorbdb4

COMPLEX for cunbdb4

DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (*q*).

Scalar factors of the elementary reflectors that define q_1.

phantom

REAL for sorbdb4

DOUBLE PRECISION for dorbdb4

COMPLEX for cunbdb4

DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (*m*).

The routine computes an *m*-by-1 column vector *y* that is orthogonal to the columns of [*x11*; *x21*]. *phantom*(1:*p*) and *phantom*(*p*+1:*m*) contain Householder vectors for *y*(1:*p*) and *y*(*p*+1:*m*), respectively.

info

INTEGER.

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument has an illegal value.

See Also

?orcsd/?uncsd

?orcsd2by1/?uncsd2by1 Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

?orbdb1/?unbdb1 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb2/?unbdb2 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb3/?unbdb3 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb5/?unbdb5 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

?orbdb6/?unbdb6 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

xerbla

?orbdb5/?unbdb5

Orthogonalizes a column vector with respect to the orthonormal basis matrix.

Syntax

```
call sorbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call dorbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call cunbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call zunbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The ?orbdb5/?unbdb5 routines orthogonalize the column vector

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

with respect to the columns of

$$q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$$

The columns of Q must be orthonormal.

If the projection is zero according to Kahan's "twice is enough" criterion, then some other vector from the orthogonal complement is returned. This vector is chosen in an arbitrary but deterministic way.

Input Parameters

m1

INTEGER

The dimension of *x1* and the number of rows in *q1*. 0 ≤*m1*.

m2

INTEGER

The dimension of *x2* and the number of rows in *q2*. 0 ≤*m2*.

n

INTEGER

The number of columns in *q1* and *q2*. 0 ≤*n*.

x1

REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Array of size *m1*.

The top part of the vector to be orthogonalized.

incx1

INTEGER

	Increment for entries of *x1*.
x2	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5 Array of size *m2*. The bottom part of the vector to be orthogonalized.
incx2	INTEGER Increment for entries of *x2*.
q1	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5 Array of size (*ldq1*, *n*). The top part of the orthonormal basis matrix.
ldq1	INTEGER The leading dimension of *q1*. *ldq1*≥*m1*.
q2	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5 Array of size (*ldq2*, *n*). The bottom part of the orthonormal basis matrix.
ldq2	INTEGER The leading dimension of *q2*. *ldq2*≥*m2*.
work	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5 Workspace array of size *lwork*.
lwork	INTEGER The size of the array *work*. *lwork*≥*n*.

Output Parameters

x1	The top part of the projected vector.

x2	The bottom part of the projected vector.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value.

See Also

?orcsd/?uncsd

?orcsd2by1/?uncsd2by1 Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

?orbdb1/?unbdb1 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb2/?unbdb2 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb3/?unbdb3 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb4/?unbdb4 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb6/?unbdb6 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

xerbla

?orbdb6/?unbdb6

Orthogonalizes a column vector with respect to the orthonormal basis matrix.

Syntax

```
call sorbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call dorbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call cunbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call zunbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The ?orbdb6/?unbdb6 routines orthogonalize the column vector

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

with respect to the columns of

$$q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$$

The columns of Q must be orthonormal.

If the projection is zero according to Kahan's "twice is enough" criterion, then the zero vector is returned.

Input Parameters

m1 INTEGER

The dimension of *x1* and the number of rows in *q1*. 0 ≤*m1*.

m2 INTEGER

The dimension of *x2* and the number of rows in *q2*. 0 ≤*m2*.

n INTEGER

The number of columns in *q1* and *q2*. 0 ≤*n*.

x1 REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Array of size *m1*.

The top part of the vector to be orthogonalized.

incx1 INTEGER

Increment for entries of *x1*.

x2 REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Array of size *m2*.

The bottom part of the vector to be orthogonalized.

incx2 INTEGER

Increment for entries of *x2*.

q1 REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Array of size (*ldq1*, *n*).

The top part of the orthonormal basis matrix.

ldq1 INTEGER

The leading dimension of *q1*. *ldq1*≥*m1*.

q2 REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Array of size (*ldq2*, *n*).

The bottom part of the orthonormal basis matrix.

ldq2 INTEGER

The leading dimension of *q2*. *ldq2*≥*m2*.

work REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Workspace array of size *lwork*.

lwork INTEGER

The size of the array *work*. *lwork*≥*n*.

Output Parameters

x1 The top part of the projected vector.

x2 The bottom part of the projected vector.

info INTEGER.

= 0: successful exit

< 0: if *info* = −*i*, the *i*-th argument has an illegal value.

See Also

?orcsd/?uncsd

?orcsd2by1/?uncsd2by1 Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

?orbdb1/?unbdb1 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb2/?unbdb2 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb3/?unbdb3 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb4/?unbdb4 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb5/?unbdb5 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

xerbla

?org2l/?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by ?geqlf *(unblocked algorithm).*

Syntax

```
call sorg2l( m, n, k, a, lda, tau, work, info )
call dorg2l( m, n, k, a, lda, tau, work, info )
```

```
call cung2l( m, n, k, a, lda, tau, work, info )
call zung2l( m, n, k, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine ?org2l/?ung2l generates an *m*-by-*n* real/complex matrix *Q* with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors of order *m*:

Q = *H*(*k*)*...*H*(2)*H*(1) as returned by ?geqlf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q*. *m*≥ 0.
n	INTEGER. The number of columns of the matrix *Q*. *m*≥*n*≥ 0.
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. *n*≥*k*≥ 0.
a	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l DOUBLE COMPLEX for zung2l. Array, DIMENSION (*lda*,*n*). On entry, the (*n*-*k*+*i*)-th column must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?geqlf in the last *k* columns of its array argument *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*m*).
tau	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l DOUBLE COMPLEX for zung2l. Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?geqlf.
work	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l DOUBLE COMPLEX for zung2l. Workspace array, DIMENSION (*n*).

Output Parameters

a	On exit, the *m*-by-*n* matrix *Q*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value

?org2r/?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by ?geqrf *(unblocked algorithm).*

Syntax

```
call sorg2r( m, n, k, a, lda, tau, work, info )
call dorg2r( m, n, k, a, lda, tau, work, info )
call cung2r( m, n, k, a, lda, tau, work, info )
call zung2r( m, n, k, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine ?org2r/?ung2r generates an *m*-by-*n* real/complex matrix *Q* with orthonormal columns, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*

Q = *H*(1)**H*(2)*...**H*(*k*)

as returned by ?geqrf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q*. *m*≥ 0.
n	INTEGER. The number of columns of the matrix *Q*. *m*≥*n*≥ 0.
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. *n*≥*k*≥ 0.
a	REAL for sorg2r DOUBLE PRECISION for dorg2r COMPLEX for cung2r DOUBLE COMPLEX for zung2r. Array, DIMENSION (*lda*, *n*). On entry, the *i*-th column must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?geqrf in the first *k* columns of its array argument *a*.
lda	INTEGER. The first DIMENSION of the array *a*. *lda*≥ max(1,*m*).

tau	REAL for sorg2r DOUBLE PRECISION for dorg2r COMPLEX for cung2r DOUBLE COMPLEX for zung2r. Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?geqrf.
work	REAL for sorg2r DOUBLE PRECISION for dorg2r COMPLEX for cung2r DOUBLE COMPLEX for zung2r. Workspace array, DIMENSION (*n*).

Output Parameters

a	On exit, the *m*-by-*n* matrix *Q*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value

?orgl2/?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by ?gelqf *(unblocked algorithm).*

Syntax

```
call sorgl2( m, n, k, a, lda, tau, work, info )
call dorgl2( m, n, k, a, lda, tau, work, info )
call cungl2( m, n, k, a, lda, tau, work, info )
call zungl2( m, n, k, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine ?orgl2/?ungl2 generates a *m*-by-*n* real/complex matrix *Q* with orthonormal rows, which is defined as the first *m* rows of a product of *k* elementary reflectors of order n

Q = *H*(*k*)*...**H*(2)**H*(1) for real flavors, or *Q* = (*H*(*k*))^H*...*(*H*(2))^H*(*H*(1))^H for complex flavors as returned by ?gelqf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q*. *m* ≥ 0.

n	INTEGER. The number of columns of the matrix *Q*. *n*≥*m*.
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. *m*≥*k*≥ 0.
a	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 DOUBLE COMPLEX for zungl2. Array, DIMENSION (*lda*, *n*). On entry, the *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?gelqf in the first *k* rows of its array argument *a*.
lda	INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*m*).
tau	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 DOUBLE COMPLEX for zungl2. Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?gelqf.
work	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 DOUBLE COMPLEX for zungl2. Workspace array, DIMENSION (*m*).

Output Parameters

a	On exit, the *m*-by-*n* matrix *Q*.
info	INTEGER. = 0: successful exit < 0: if *info* = −*i*, the *i*-th argument has an illegal value.

?orgr2/?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf *(unblocked algorithm).*

Syntax

```
call sorgr2( m, n, k, a, lda, tau, work, info )
call dorgr2( m, n, k, a, lda, tau, work, info )
call cungr2( m, n, k, a, lda, tau, work, info )
```

```
call zungr2( m, n, k, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine ?orgr2/?ungr2 generates an *m*-by-*n* real matrix *Q* with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *n*

$Q = H(1)*H(2)*...*H(k)$ for real flavors, or $Q = (H(1))^H*(H(2))^H*...*(H(k))^H$ for complex flavors as returned by ?gerqf.

Input Parameters

m — INTEGER. The number of rows of the matrix *Q*. $m \geq 0$.

n — INTEGER. The number of columns of the matrix *Q*. $n \geq m$

k — INTEGER.

The number of elementary reflectors whose product defines the matrix *Q*. $m \geq k \geq 0$.

a — REAL for sorgr2

DOUBLE PRECISION for dorgr2

COMPLEX for cungr2

DOUBLE COMPLEX for zungr2.

Array, DIMENSION (*lda*, *n*).

On entry, the (*m*- *k*+i)-th row must contain the vector which defines the elementary reflector *H*(*i*), for $i = 1,2,...,k$, as returned by ?gerqf in the last *k* rows of its array argument *a*.

lda — INTEGER. The leading dimension of the array *a*. $lda \geq \max(1,m)$.

tau — REAL for sorgr2

DOUBLE PRECISION for dorgr2

COMPLEX for cungr2

DOUBLE COMPLEX for zungr2.

Array, DIMENSION (*k*).*tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?gerqf.

work — REAL for sorgr2

DOUBLE PRECISION for dorgr2

COMPLEX for cungr2

DOUBLE COMPLEX for zungr2.

Workspace array, DIMENSION (*m*).

Output Parameters

a	On exit, the *m*-by-*n* matrix *Q*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value

?orm2l/?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf *(unblocked algorithm).*

Syntax

```
call sorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- mkl.fi

Description

The routine ?orm2l/?unm2l overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q***C* if *side* = 'L' and *trans* = 'N', or

Q^T**C* / Q^H**C* if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

*C***Q* if *side* = 'R' and *trans* = 'N', or

*C**Q^T / *C**Q^H if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

Here *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(*k*)*...**H*(2)**H*(1) as returned by ?geqlf.

Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	CHARACTER*1. = 'L': apply *Q* or Q^T / Q^H from the left = 'R': apply *Q* or Q^T / Q^H from the right
trans	CHARACTER*1. = 'N': apply *Q* (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
m	INTEGER. The number of rows of the matrix *C*. *m* ≥ 0.

n INTEGER. The number of columns of the matrix *C*. *n*≥ 0.

k INTEGER. The number of elementary reflectors whose product defines the matrix *Q*.

If *side* = 'L', *m*≥*k*≥ 0;

if *side* = 'R', *n*≥*k*≥ 0.

a REAL for sorm2l

DOUBLE PRECISION for dorm2l

COMPLEX for cunm2l

DOUBLE COMPLEX for zunm2l.

Array, DIMENSION (*lda*,*k*).

The *i*-th column must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?geqlf in the last *k* columns of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda INTEGER. The leading dimension of the array *a*.

If *side* = 'L', *lda*≥ max(1, *m*)

if *side* = 'R', *lda*≥ max(1, *n*).

tau REAL for sorm2l

DOUBLE PRECISION for dorm2l

COMPLEX for cunm2l

DOUBLE COMPLEX for zunm2l.

Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?geqlf.

c REAL for sorm2l

DOUBLE PRECISION for dorm2l

COMPLEX for cunm2l

DOUBLE COMPLEX for zunm2l.

Array, DIMENSION (*ldc*, *n*).

On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *C*. *ldc*≥ max(1,*m*).

work REAL for sorm2l

DOUBLE PRECISION for dorm2l

COMPLEX for cunm2l

DOUBLE COMPLEX for zunm2l.

Workspace array, DIMENSION:

(*n*) if *side* = 'L',

(*m*) if *side* = 'R'.

Output Parameters

c — On exit, *c* is overwritten by *Q*C* or Q^T*C / Q^H*C, or *C*Q*, or *C*Q^T* / *C*Q^H*.

info — INTEGER.

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument had an illegal value

?orm2r/?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf *(unblocked algorithm).*

Syntax

```
call sorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- mkl.fi

Description

The routine ?orm2r/?unm2r overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q*C* if *side* = 'L' and *trans* = 'N', or

Q^T*C / Q^H*C if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

*C*Q* if *side* = 'R' and *trans* = 'N', or

CQ^T* / *C*Q^H* if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

Here *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(1)**H*(2)*...**H*(*k*) as returned by ?geqrf.

Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — CHARACTER*1.

= 'L': apply *Q* or Q^T / Q^H from the left

= 'R': apply *Q* or Q^T / Q^H from the right

trans — CHARACTER*1.

= 'N': apply *Q* (no transpose)

= 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H (conjugate transpose, for complex flavors)

m INTEGER. The number of rows of the matrix *C*. *m*≥ 0.

n INTEGER. The number of columns of the matrix *C*. *n*≥ 0.

k INTEGER. The number of elementary reflectors whose product defines the matrix *Q*.

If *side* = 'L', *m*≥*k*≥ 0;

if *side* = 'R', *n*≥*k*≥ 0.

a REAL for sorm2r

DOUBLE PRECISION for dorm2r

COMPLEX for cunm2r

DOUBLE COMPLEX for zunm2r.

Array, DIMENSION (*lda*,*k*).

The *i*-th column must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?geqrf in the first *k* columns of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda INTEGER. The leading dimension of the array *a*.

If *side* = 'L', *lda*≥ max(1, *m*);

if *side* = 'R', *lda*≥ max(1, *n*).

tau REAL for sorm2r

DOUBLE PRECISION for dorm2r

COMPLEX for cunm2r

DOUBLE COMPLEX for zunm2r.

Array, DIMENSION (*k*).

tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?geqrf.

c REAL for sorm2r

DOUBLE PRECISION for dorm2r

COMPLEX for cunm2r

DOUBLE COMPLEX for zunm2r.

Array, DIMENSION (*ldc*, *n*).

On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*. *ldc*≥ max(1,*m*).

work REAL for sorm2r

DOUBLE PRECISION for dorm2r

COMPLEX for cunm2r

DOUBLE COMPLEX for zunm2r.

Workspace array, DIMENSION

(*n*) if *side* = 'L',

(*m*) if *side* = 'R'.

Output Parameters

c — On exit, *c* is overwritten by *Q*C* or Q^T*C / Q^H*C, or *C*Q*, or *C*Q^T* / *C*Q^H*.

info — INTEGER.

= 0: successful exit

< 0: if *info* = -i, the *i*-th argument had an illegal value

?orml2/?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf *(unblocked algorithm).*

Syntax

```
call sorml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- mkl.fi

Description

The routine ?orml2/?unml2 overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q*C* if *side* = 'L' and *trans* = 'N', or

Q^T*C / Q^H*C if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

*C*Q* if *side* = 'R' and *trans* = 'N', or

CQ^T* / *C*Q^H* if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

Here *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(*k*)*...**H*(2)**H*(1) for real flavors, or *Q* = (*H*(*k*))^H*...*(*H*(2))^H*(*H*(1))^H for complex flavors as returned by ?gelqf.

Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — CHARACTER*1.

= 'L': apply *Q* or Q^T / Q^H from the left

= 'R': apply *Q* or Q^T / Q^H from the right

trans — CHARACTER*1.

= 'N': apply Q (no transpose)

= 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H (conjugate transpose, for complex flavors)

m INTEGER. The number of rows of the matrix *C*. $m \geq 0$.

n INTEGER. The number of columns of the matrix *C*. $n \geq 0$.

k INTEGER. The number of elementary reflectors whose product defines the matrix *Q*.

If *side* = 'L', $m \geq k \geq 0$;

if *side* = 'R', $n \geq k \geq 0$.

a REAL for sorml2

DOUBLE PRECISION for dorml2

COMPLEX for cunml2

DOUBLE COMPLEX for zunml2.

Array, DIMENSION

(*lda*, *m*) if *side* = 'L',

(*lda*, *n*) if *side* = 'R'

The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?gelqf in the first *k* rows of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, k)$.

tau REAL for sorml2

DOUBLE PRECISION for dorml2

COMPLEX for cunml2

DOUBLE COMPLEX for zunml2.

Array, DIMENSION (*k*).

tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?gelqf.

c REAL for sorml2

DOUBLE PRECISION for dorml2

COMPLEX for cunml2

DOUBLE COMPLEX for zunml2.

Array, DIMENSION (*ldc*, *n*) On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*. $ldc \geq \max(1, m)$.

work REAL for sorml2

DOUBLE PRECISION for dorml2

COMPLEX for cunml2

DOUBLE COMPLEX for zunml2.

Workspace array, DIMENSION

(*n*) if *side* = 'L',

(*m*) if *side* = 'R'

Output Parameters

c — On exit, *c* is overwritten by *Q*C* or Q^T*C / Q^H*C, or *C*Q*, or $C*Q^T$ / $C*Q^H$.

info — INTEGER.

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument had an illegal value

?ormr2/?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf *(unblocked algorithm).*

Syntax

```
call sormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- mkl.fi

Description

The routine ?ormr2/?unmr2 overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q*C* if *side* = 'L' and *trans* = 'N', or

Q^T*C / Q^H*C if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

*C*Q* if *side* = 'R' and *trans* = 'N', or

$C*Q^T$ / $C*Q^H$ if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

Here *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(1)**H*(2)*...**H*(*k*) for real flavors, or *Q* = $(H(1))^H*(H(2))^H*...*(H(k))^H$ as returned by ?gerqf.

Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — CHARACTER*1.

= 'L': apply *Q* or Q^T / Q^H from the left

= 'R': apply *Q* or Q^T / Q^H from the right

trans CHARACTER*1.

= 'N': apply Q (no transpose)

= 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H (conjugate transpose, for complex flavors)

m INTEGER. The number of rows of the matrix C. $m \geq 0$.

n INTEGER. The number of columns of the matrix C. $n \geq 0$.

k INTEGER. The number of elementary reflectors whose product defines the matrix Q.

If *side* = 'L', $m \geq k \geq 0$;

if *side* = 'R', $n \geq k \geq 0$.

a REAL for sormr2

DOUBLE PRECISION for dormr2

COMPLEX for cunmr2

DOUBLE COMPLEX for zunmr2.

Array, DIMENSION

(*lda*, *m*) if *side* = 'L',

(*lda*, *n*) if *side* = 'R'

The *i*-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1,2,\ldots,k$, as returned by ?gerqf in the last k rows of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda INTEGER.

The leading dimension of the array *a*. $lda \geq \max(1,k)$.

tau REAL for sormr2

DOUBLE PRECISION for dormr2

COMPLEX for cunmr2

DOUBLE COMPLEX for zunmr2.

Array, DIMENSION (*k*).

tau(*i*) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gerqf.

c REAL for sormr2

DOUBLE PRECISION for dormr2

COMPLEX for cunmr2

DOUBLE COMPLEX for zunmr2.

Array, DIMENSION (*ldc*, *n*).

On entry, the m-by-n matrix C.

ldc	INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1,*m*).
work	REAL for sormr2 DOUBLE PRECISION for dormr2 COMPLEX for cunmr2 DOUBLE COMPLEX for zunmr2. Workspace array, DIMENSION (*n*) if *side* = 'L', (*m*) if *side* = 'R'

Output Parameters

c	On exit, *c* is overwritten by *Q*C* or Q^T*C / Q^H*C, or *C*Q*, or $C*Q^T$ / $C*Q^H$.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value

?ormr3/?unmr3

Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzrzf *(unblocked algorithm).*

Syntax

```
call sormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call dormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call cunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call zunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
```

Include Files

- mkl.fi

Description

The routine ?ormr3/?unmr3 overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q*C* if *side* = 'L' and *trans* = 'N', or

Q^T*C / Q^H*C if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

*C*Q* if *side* = 'R' and *trans* = 'N', or

$C*Q^T$ / $C*Q^H$ if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

Here *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(1)**H*(2)*...**H*(*k*) as returned by ?tzrzf.

Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side CHARACTER*1.

= 'L': apply Q or Q^T / Q^H from the left

= 'R': apply Q or Q^T / Q^H from the right

trans CHARACTER*1.

= 'N': apply Q (no transpose)

= 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H (conjugate transpose, for complex flavors)

m INTEGER. The number of rows of the matrix C. $m \geq 0$.

n INTEGER. The number of columns of the matrix C. $n \geq 0$.

k INTEGER. The number of elementary reflectors whose product defines the matrix Q.

If *side* = 'L', $m \geq k \geq 0$;

if *side* = 'R', $n \geq k \geq 0$.

l INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors.

If *side* = 'L', $m \geq l \geq 0$,

if *side* = 'R', $n \geq l \geq 0$.

a REAL for sormr3

DOUBLE PRECISION for dormr3

COMPLEX for cunmr3

DOUBLE COMPLEX for zunmr3.

Array, DIMENSION

(*lda*, *m*) if *side* = 'L',

(*lda*, *n*) if *side* = 'R'

The i-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \ldots, k$, as returned by ?tzrzf in the last k rows of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda INTEGER.

The leading dimension of the array *a*. $lda \geq \max(1, k)$.

tau REAL for sormr3

DOUBLE PRECISION for dormr3

COMPLEX for cunmr3

DOUBLE COMPLEX for zunmr3.

Array, DIMENSION (k).

tau(*i*) must contain the scalar factor of the elementary reflector H(*i*), as returned by `?tzrzf`.

c — `REAL` for `sormr3`
`DOUBLE PRECISION` for `dormr3`
`COMPLEX` for `cunmr3`
`DOUBLE COMPLEX` for `zunmr3`.
Array, `DIMENSION` (*ldc*, *n*).
On entry, the *m*-by-*n* matrix *C*.

ldc — `INTEGER`. The leading dimension of the array *c*. *ldc*≥ `max(1,m)`.

work — `REAL` for `sormr3`
`DOUBLE PRECISION` for `dormr3`
`COMPLEX` for `cunmr3`
`DOUBLE COMPLEX` for `zunmr3`.
Workspace array, `DIMENSION`
(*n*) if *side* = `'L'`,
(*m*) if *side* = `'R'`.

Output Parameters

c — On exit, *c* is overwritten by $Q*C$ or Q^T*C / Q^H*C, or $C*Q$, or $C*Q^T$ / $C*Q^H$.

info — `INTEGER`.
= 0: successful exit
< 0: if *info* = *-i*, the *i*-th argument had an illegal value

?pbtf2

Computes the Cholesky factorization of a symmetric/ Hermitian positive-definite band matrix (unblocked algorithm).

Syntax

```
call spbtf2( uplo, n, kd, ab, ldab, info )
call dpbtf2( uplo, n, kd, ab, ldab, info )
call cpbtf2( uplo, n, kd, ab, ldab, info )
call zpbtf2( uplo, n, kd, ab, ldab, info )
```

Include Files

- `mkl.fi`

Description

The routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite band matrix *A*.

The factorization has the form

$A = U^T*U$ for real flavors, $A = U^H*U$ for complex flavors if *uplo* = 'U', or

$A = L*L^T$ for real flavors, $A = L*L^H$ for complex flavors if *uplo* = 'L',

where *U* is an upper triangular matrix, and *L* is lower triangular. This is the unblocked version of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo — CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix *A* is stored:

= 'U': upper triangular

= 'L': lower triangular

n — INTEGER. The order of the matrix *A*. $n \geq 0$.

kd — INTEGER. The number of super-diagonals of the matrix *A* if *uplo* = 'U', or the number of sub-diagonals if *uplo* = 'L'.

$kd \geq 0$.

ab — REAL for spbtf2

DOUBLE PRECISION for dpbtf2

COMPLEX for cpbtf2

DOUBLE COMPLEX for zpbtf2.

Array, DIMENSION (*ldab*, *n*).

On entry, the upper or lower triangle of the symmetric/ Hermitian band matrix *A*, stored in the first *kd*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

if *uplo* = 'U', $ab(kd+1+i-j,j) = A(i, j)$ for $\max(1, j-kd) \leq i \leq j$;

if *uplo* = 'L', $ab(1+i-j,j) = A(i, j)$ for $j \leq i \leq \min(n, j+kd)$.

ldab — INTEGER. The leading dimension of the array *ab*. $ldab \geq kd+1$.

Output Parameters

ab — On exit, If *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization $A=U^T*U$ ($A=U^H*U$), or $A= L*L^T$ ($A = L*L^H$) of the band matrix *A*, in the same storage format as *A*.

info — INTEGER.

= 0: successful exit

< 0: if *info* = −*k*, the *k*-th argument had an illegal value

> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

?potf2

Computes the Cholesky factorization of a symmetric/ Hermitian positive-definite matrix (unblocked algorithm).

Syntax

```
call spotf2( uplo, n, a, lda, info )
call dpotf2( uplo, n, a, lda, info )
call cpotf2( uplo, n, a, lda, info )
call zpotf2( uplo, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine ?potf2 computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite matrix *A*. The factorization has the form

$A = U^T*U$ for real flavors, $A = U^H*U$ for complex flavors if *uplo* = 'U', or

$A = L*L^T$ for real flavors, $A = L*L^H$ for complex flavors if *uplo* = 'L',

where *U* is an upper triangular matrix, and *L* is lower triangular.

This is the unblocked version of the algorithm, calling BLAS Level 2 Routines

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix *A* is stored.

= 'U': upper triangular

= 'L': lower triangular

n

INTEGER. The order of the matrix *A*. $n \geq 0$.

a

REAL for spotf2

DOUBLE PRECISION or dpotf2

COMPLEX for cpotf2

DOUBLE COMPLEX for zpotf2.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric/Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*.

lda ≥ max(1,*n*).

Output Parameters

a On exit, If *info* = 0, the factor *U* or *L* from the Cholesky factorization $A=U^{T}*U$ ($A=U^{H}*U$), or $A= L*L^{T}$ ($A = L*L^{H}$).

info INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument had an illegal value

> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

?ptts2

Solves a tridiagonal system of the form $A*X=B$ *using the* $L*D*L^{H}/L*D*L^{H}$ *factorization computed by* ?pttrf.

Syntax

```
call sptts2( n, nrhs, d, e, b, ldb )
call dptts2( n, nrhs, d, e, b, ldb )
call cptts2( iuplo, n, nrhs, d, e, b, ldb )
call zptts2( iuplo, n, nrhs, d, e, b, ldb )
```

Include Files

- mkl.fi

Description

The routine ?ptts2 solves a tridiagonal system of the form

$A*X = B$

Real flavors sptts2/dptts2 use the $L*D*L^{T}$ factorization of *A* computed by spttrf/dpttrf, and complex flavors cptts2/zptts2 use the $U^{H}*D*U$ or $L*D*L^{H}$ factorization of *A* computed by cpttrf/zpttrf.

D is a diagonal matrix specified in the vector *d*, *U* (or *L*) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector *e*, and *X* and *B* are *n*-by-*nrhs* matrices.

Input Parameters

iuplo INTEGER. Used with complex flavors only.

Specifies the form of the factorization, and whether the vector *e* is the superdiagonal of the upper bidiagonal factor *U* or the subdiagonal of the lower bidiagonal factor *L*.

= 1: $A = U^{H}*D*U$, *e* is the superdiagonal of *U*;

= 0: $A = L*D*L^{H}$, *e* is the subdiagonal of *L*

n INTEGER. The order of the tridiagonal matrix *A*. *n* ≥ 0.

nrhs INTEGER. The number of right hand sides, that is, the number of columns of the matrix *B*. nrhs≥ 0.

d REAL for sptts2/cptts2

DOUBLE PRECISION for dptts2/zptts2.

Array, DIMENSION (*n*).

The *n* diagonal elements of the diagonal matrix *D* from the factorization of *A*.

e REAL for sptts2

DOUBLE PRECISION for dptts2

COMPLEX for cptts2

DOUBLE COMPLEX for zptts2.

Array, DIMENSION (*n*-1).

Contains the (*n*-1) subdiagonal elements of the unit bidiagonal factor *L* from the $L*D*L^T$ (for real flavors) or $L*D*L^H$ (for complex flavors when *iuplo* = 0) factorization of *A*.

For complex flavors when *iuplo* = 1, *e* contains the (*n*-1) superdiagonal elements of the unit bidiagonal factor *U* from the factorization $A = U^H*D*U$.

B REAL for sptts2/cptts2

DOUBLE PRECISION for dptts2/zptts2.

Array, DIMENSION (*ldb*, *nrhs*).

On entry, the right hand side vectors *B* for the system of linear equations.

ldb INTEGER. The leading dimension of the array *B*. *ldb*≥ max(1,*n*).

Output Parameters

b On exit, the solution vectors, *X*.

?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call srscl( n, sa, sx, incx )
call drscl( n, sa, sx, incx )
call csrscl( n, sa, sx, incx )
call zdrscl( n, sa, sx, incx )
```

Include Files

- mkl.fi

Description

The routine ?rscl multiplies an *n*-element real/complex vector *x* by the real scalar 1/*a*. This is done without overflow or underflow as long as the final result *x*/*a* does not overflow or underflow.

Input Parameters

n	INTEGER. The number of components of the vector *x*.
sa	REAL for srscl/csrscl DOUBLE PRECISION for drscl/zdrscl. The scalar a which is used to divide each component of the vector *x*. sa must be ≥ 0, or the subroutine will divide by zero.
sx	REAL for srscl DOUBLE PRECISION for drscl COMPLEX for csrscl DOUBLE COMPLEX for zdrscl. Array, DIMENSION(1+(*n*-1)*|*incx*|). The *n*-element vector *x*.
incx	INTEGER. The increment between successive values of the vector *sx*. If *incx* > 0, *sx*(1)=*x*(1), and *sx*(1+(i-1)**incx*)=*x*(*i*), 1<*i*≤*n*.

Output Parameters

sx	On exit, the result *x*/*a*.

?syswapr

Applies an elementary permutation on the rows and columns of a symmetric matrix.

Syntax

```
call ssyswapr( uplo, n, a, lda, i1, i2 )
call dsyswapr( uplo, n, a, lda, i1, i2 )
call csyswapr( uplo, n, a, lda, i1, i2 )
call zsyswapr( uplo, n, a, lda, i1, i2 )
call syswapr( a,i1,i2[,uplo] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine applies an elementary permutation on the rows and columns of a symmetric matrix.

Input Parameters

The data types are given for the Fortran interface.

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization $A = U*D*U^T$.

If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization $A = L*D*L^T$.

n — INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a — REAL for ssyswapr

DOUBLE PRECISION for dsyswapr

COMPLEX for csyswapr

DOUBLE COMPLEX for zsyswapr

Array of size (*lda*, *n*).

The array *a* contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* as computed by ?sytrf.

lda — INTEGER. The leading dimension of the array *a*. $lda \geq max(1,n)$.

i1 — INTEGER. Index of the first row to swap.

i2 — INTEGER. Index of the second row to swap.

Output Parameters

a

If *info* = 0, the symmetric inverse of the original matrix.

If *info* = 'U', the upper triangular part of the inverse is formed and the part of *A* below the diagonal is not referenced.

If *info* = 'L', the lower triangular part of the inverse is formed and the part of *A* above the diagonal is not referenced.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine syswapr interface are as follows:

a — Holds the matrix *A* of size (*n*, *n*).

i1 — Holds the index for swap.

i2 — Holds the index for swap.

uplo — Indicates how the matrix *A* has been factored. Must be 'U' or 'L'.

See Also

?sytrf

?heswapr

Applies an elementary permutation on the rows and columns of a Hermitian matrix.

Syntax

```
call cheswapr( uplo, n, a, lda, i1, i2 )
call zheswapr( uplo, n, a, lda, i1, i2 )
call heswapr( a, i1, i2 [,uplo] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine applies an elementary permutation on the rows and columns of a Hermitian matrix.

Input Parameters

The data types are given for the Fortran interface.

uplo — CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization $A = U*D*U^H$.

If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization $A = L*D*L^H$.

n — INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a — COMPLEX for cheswapr

DOUBLE COMPLEX for zheswapr

Array of size (*lda*, *n*).

The array *a* contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* as computed by ?hetrf.

lda — INTEGER. The leading dimension of the array *a*. $lda \geq max(1,n)$.

i1 — INTEGER. Index of the first row to swap.

i2 — INTEGER. Index of the second row to swap.

Output Parameters

a — If *info* = 0, the inverse of the original matrix.

If *info* = 'U', the upper triangular part of the inverse is formed and the part of *A* below the diagonal is not referenced.

If *info* = 'L', the lower triangular part of the inverse is formed and the part of *A* above the diagonal is not referenced.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `heswapr` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
i1	Holds the index for swap.
i2	Holds the index for swap.
uplo	Must be `'U'` or `'L'`.

See Also

?hetrf

?syswapr1

?syswapr1

Applies an elementary permutation on the rows and columns of a symmetric matrix.

Syntax

```
call ssyswapr1( uplo, n, a, lda, i1, i2 )
call dsyswapr1( uplo, n, a, lda, i1, i2 )
call csyswapr1( uplo, n, a, lda, i1, i2 )
call zsyswapr1( uplo, n, a, lda, i1, i2 )
call syswapr1( a,i1,i2[,uplo] )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine applies an elementary permutation on the rows and columns of a symmetric matrix.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates how the input matrix *A* has been factored: If *uplo* = `'U'`, the array *a* stores the upper triangular factor *U* of the factorization *A* = $U*D*U^T$. If *uplo* = `'L'`, the array *a* stores the lower triangular factor *L* of the factorization *A* = $L*D*L^T$.
n	`INTEGER`. The order of matrix *A*; $n \geq 0$.
nrhs	`INTEGER`. The number of right-hand sides; $nrhs \geq 0$.

a	`REAL` for `ssyswapr1`
	`DOUBLE PRECISION` for `dsyswapr1`
	`COMPLEX` for `csyswapr1`
	`DOUBLE COMPLEX` for `zsyswapr1`
	Array of dimension `(lda,n)`.

The array *a* contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* as computed by `?sytrf`.

lda — `INTEGER`. The leading dimension of the array `a`. *lda*≥*max*(1,*n*).

i1 — `INTEGER`. Index of the first row to swap.

i2 — `INTEGER`. Index of the second row to swap.

Output Parameters

a — If *info* = 0, the symmetric inverse of the original matrix.

If *info* = `'U'`, the upper triangular part of the inverse is formed and the part of *A* below the diagonal is not referenced.

If *info* = `'L'`, the lower triangular part of the inverse is formed and the part of *A* above the diagonal is not referenced.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see LAPACK 95 Interface Conventions.

Specific details for the routine `syswapr1` interface are as follows:

a — Holds the matrix *A* of size (*n*, *n*).

i1 — Holds the index for swap.

i2 — Holds the index for swap.

uplo — Indicates how the matrix *A* has been factored. Must be `'U'` or `'L'`.

See Also

`?sytrf`

?sygs2/?hegs2

Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from `?potrf` *(unblocked algorithm).*

Syntax

```
call ssygs2( itype, uplo, n, a, lda, b, ldb, info )
call dsygs2( itype, uplo, n, a, lda, b, ldb, info )
call chegs2( itype, uplo, n, a, lda, b, ldb, info )
call zhegs2( itype, uplo, n, a, lda, b, ldb, info )
```

Include Files

- `mkl.fi`

Description

The routine `?sygs2`/`?hegs2` reduces a real symmetric-definite or a complex Hermitian positive-definite generalized eigenproblem to standard form.

If *itype* = 1, the problem is

$A*x = \lambda*B*x$

and *A* is overwritten by $\text{inv}(U^H)*A*\text{inv}(U)$ or $\text{inv}(L)*A*\text{inv}(L^H)$ for complex flavors and by $\text{inv}(U^T)*A*\text{inv}(U)$ or $\text{inv}(L)*A*\text{inv}(L^T)$ for real flavors.

If *itype* = 2 or 3, the problem is

$A*B*x = \lambda*x$, or $B*A*x = \lambda*x$,

and *A* is overwritten by $U*A*U^H$ or L^H*A*L for complex flavors and by $U*A*U^T$ or L^T*A*L for real flavors. Here U^T and L^T are the transpose while U^H and L^H are conjugate transpose of *U* and *L*.

B must be previously factorized by ?potrf as follows:

- U^H*U or $L*L^H$ for complex flavors
- U^T*U or $L*L^T$ for real flavors

Input Parameters

itype

INTEGER.

For complex flavors:

= 1: compute $\text{inv}(U^H)*A*\text{inv}(U)$ or $\text{inv}(L)*A*\text{inv}(L^H)$;

= 2 or 3: compute $U*A*U^H$ or L^H*A*L.

For real flavors:

= 1: compute $\text{inv}(U^T)*A*\text{inv}(U)$ or $\text{inv}(L)*A*\text{inv}(L^T)$;

= 2 or 3: compute $U*A*U^T$ or L^T*A*L.

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored, and how *B* has been factorized.

= 'U': upper triangular

= 'L': lower triangular

n

INTEGER. The order of the matrices *A* and *B*. $n \geq 0$.

a

REAL for ssygs2

DOUBLE PRECISION for dsygs2

COMPLEX for chegs2

DOUBLE COMPLEX for zhegs2.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric/Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda

INTEGER.

The leading dimension of the array *a*. *lda*≥ max(1,*n*).

b

REAL for ssygs2

DOUBLE PRECISION for dsygs2

COMPLEX for chegs2

DOUBLE COMPLEX for zhegs2.

Array, DIMENSION (*ldb*, *n*).

The triangular factor from the Cholesky factorization of *B* as returned by ?potrf.

ldb

INTEGER. The leading dimension of the array *b*. *ldb*≥ max(1,*n*).

Output Parameters

a

On exit, If *info* = 0, the transformed matrix, stored in the same format as *A*.

info

INTEGER.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?sytd2/?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation(unblocked algorithm).

Syntax

```
call ssytd2( uplo, n, a, lda, d, e, tau, info )
call dsytd2( uplo, n, a, lda, d, e, tau, info )
call chetd2( uplo, n, a, lda, d, e, tau, info )
call zhetd2( uplo, n, a, lda, d, e, tau, info )
```

Include Files

- mkl.fi

Description

The routine ?sytd2/?hetd2 reduces a real symmetric/complex Hermitian matrix *A* to real symmetric tridiagonal form *T* by an orthogonal/unitary similarity transformation: $Q^T*A*Q = T$ ($Q^H*A*Q = T$).

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix *A* is stored:

= 'U': upper triangular

= 'L': lower triangular

n INTEGER. The order of the matrix *A*. *n* ≥ 0.

a REAL for ssytd2

DOUBLE PRECISION for dsytd2

COMPLEX for chetd2

DOUBLE COMPLEX for zhetd2.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric/Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1, *n*).

Output Parameters

a On exit, if *uplo* = 'U', the diagonal and first superdiagonal of *a* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;

if *uplo* = 'L', the diagonal and first subdiagonal of *a* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

d REAL for ssytd2/chetd2

DOUBLE PRECISION for dsytd2/zhetd2.

Array, DIMENSION (*n*).

The diagonal elements of the tridiagonal matrix *T*:

d(*i*) = *a*(*i*,*i*).

e REAL for ssytd2/chetd2

DOUBLE PRECISION for dsytd2/zhetd2.

Array, DIMENSION (*n*-1).

The off-diagonal elements of the tridiagonal matrix *T*:

e(*i*) = *a*(*i*,*i*+1) if *uplo* = 'U',

e(*i*) = *a*(*i*+1,*i*) if *uplo* = 'L'.

tau REAL for ssytd2

DOUBLE PRECISION for dsytd2

COMPLEX for chetd2

DOUBLE COMPLEX for zhetd2.

Array, DIMENSION (*n*).

The first *n*-1 elements contain scalar factors of the elementary reflectors. *tau*(*n*) is used as workspace.

info INTEGER.

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?sytf2

Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call ssytf2( uplo, n, a, lda, ipiv, info )
call dsytf2( uplo, n, a, lda, ipiv, info )
call csytf2( uplo, n, a, lda, ipiv, info )
call zsytf2( uplo, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine ?sytf2 computes the factorization of a real/complex symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method:

$A = U*D*U^T$, or $A = L*D*L^T$,

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored

= 'U': upper triangular

= 'L': lower triangular

n INTEGER. The order of the matrix *A*. $n \geq 0$.

a REAL for ssytf2

DOUBLE PRECISION for dsytf2

COMPLEX for csytf2

DOUBLE COMPLEX for zsytf2.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER.

The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv INTEGER.

Array, DIMENSION (*n*).

Details of the interchanges and the block structure of *D*

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) are interchanged and *D*(*k*,*k*) is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*–1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) are interchanged and *D*(*k* - 1:*k*, *k* - 1:*k*) is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1)< 0, then rows and columns *k*+1 and -*ipiv*(*k*) were interchanged and *D*(*k*:*k* + 1,*k*:*k* + 1) is a 2-by-2 diagonal block.

info INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument has an illegal value

> 0: if *info* = *k*, *D*(*k*,*k*) is exactly zero. The factorization are completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?sytf2_rook

Computes the factorization of a real/complex symmetric indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).

Syntax

```
call ssytf2_rook( uplo, n, a, lda, ipiv, info )
call dsytf2_rook( uplo, n, a, lda, ipiv, info )
call csytf2_rook( uplo, n, a, lda, ipiv, info )
call zsytf2_rook( uplo, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine ?sytf2_rook computes the factorization of a real/complex symmetric matrix *A* using the bounded Bunch-Kaufman ("rook") diagonal pivoting method:

A = *U***D***U*^{T}, or *A* = *L***D***L*^{T},

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored

= 'U': upper triangular

= 'L': lower triangular

n

INTEGER. The order of the matrix *A*. *n* ≥ 0.

a

REAL for ssytf2_rook

DOUBLE PRECISION for dsytf2_rook

COMPLEX for csytf2_rook

DOUBLE COMPLEX for zsytf2_rook.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda

INTEGER.

The leading dimension of the array *a*. *lda* ≥ max(1, *n*).

Output Parameters

a

On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv

INTEGER.

Array, DIMENSION (*n*).

Details of the interchanges and the block structure of *D*

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) < 0 and *ipiv*(*k* – 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* - 1 and -*ipiv*(*k* - 1) were interchanged, and $D_{k-1:k,\ k-1:k}$ is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) < 0 and *ipiv*(*k* + 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* + 1 and -*ipiv*(*k* + 1) were interchanged, and $D_{k:k+1,\ k:k+1}$ is a 2-by-2 diagonal block.

info INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument has an illegal value

> 0: if *info* = *k*, *D*(*k*,*k*) is exactly zero. The factorization are completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hetf2

Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call chetf2( uplo, n, a, lda, ipiv, info )
call zhetf2( uplo, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine computes the factorization of a complex Hermitian matrix *A* using the Bunch-Kaufman diagonal pivoting method:

$A = U*D*U^H$ or $A = L*D*L^H$

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, U^H is the conjugate transpose of *U*, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo CHARACTER*1.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n INTEGER. The order of the matrix *A*. $n \geq 0$.

A

COMPLEX for chetf2

DOUBLE COMPLEX for zhetf2.

Array, DIMENSION (*lda*, *n*).

On entry, the Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda

INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a

On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv

INTEGER. Array, DIMENSION (*n*).

Details of the interchanges and the block structure of *D*

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and *D(k,k)* is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*–1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) were interchanged and *D(k-1:k,k-1:k)* is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, then rows and columns *k*+1 and -*ipiv*(*k*) were interchanged and *D(k:k*+1, *k:k*+1) is a 2-by-2 diagonal block.

info

INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument had an illegal value

> 0: if *info* = *k*, *D(k,k)* is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hetf2_rook

Computes the factorization of a complex Hermitian matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).

Syntax

```
call chetf2_rook( uplo, n, a, lda, ipiv, info )
call zhetf2_rook( uplo, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine computes the factorization of a complex Hermitian matrix *A* using the bounded Bunch-Kaufman ("rook") diagonal pivoting method:

$A = U*D*U^H$ or $A = L*D*L^H$

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, U^H is the conjugate transpose of *U*, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo — CHARACTER*1.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n — INTEGER. The order of the matrix *A*. $n \geq 0$.

a — COMPLEX for chetf2_rook

DOUBLE COMPLEX for zhetf2_rook.

Array, DIMENSION (*lda*, *n*).

On entry, the Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda — INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a — On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv — INTEGER. Array, DIMENSION (*n*).

Details of the interchanges and the block structure of *D*.

If $ipiv(k) > 0$, then rows and columns *k* and *ipiv*(*k*) were interchanged and *D(k,k)* is a 1-by-1 diagonal block.

If *uplo* = 'U' and $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* - 1 and -*ipiv*(*k* - 1) were interchanged, and $D_{k-1:k,\ k-1:k}$ is a 2-by-2 diagonal block.

If *uplo* = 'L' and $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* + 1 and -*ipiv*(*k* + 1) were interchanged, and $D_{k:k+1,\ k:k+1}$ is a 2-by-2 diagonal block.

info INTEGER.

= 0: successful exit

< 0: if *info* = *-k*, the *k*-th argument had an illegal value

> 0: if *info* = *k*, *D(k,k)* is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?tgex2

Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.

Syntax

```
call stgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2, work, lwork, info )
call dtgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2, work, lwork, info )
call ctgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
call ztgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
```

Include Files

- mkl.fi

Description

The real routines stgex2/dtgex2 swap adjacent diagonal blocks (*A*11, *B*11) and (*A*22, *B*22) of size 1-by-1 or 2-by-2 in an upper (quasi) triangular matrix pair (*A*, *B*) by an orthogonal equivalence transformation. (*A*, *B*) must be in generalized real Schur canonical form (as returned by sgges/dgges), that is, *A* is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. *B* is upper triangular.

The complex routines ctgex2/ztgex2 swap adjacent diagonal 1-by-1 blocks (*A*11, *B*11) and (*A*22, *B*22) in an upper triangular matrix pair (*A*, *B*) by an unitary equivalence transformation.

(*A*, *B*) must be in generalized Schur canonical form, that is, *A* and *B* are both upper triangular.

All routines optionally update the matrices *Q* and *Z* of generalized Schur vectors:

For real flavors,

$Q(\text{in}) * A(\text{in}) * Z(\text{in})^T = Q(\text{out}) * A(\text{out}) * Z(\text{out})^T$

$Q(\text{in}) * B(\text{in}) * Z(\text{in})^T = Q(\text{out}) * B(\text{out}) * Z(\text{out})^T$.

For complex flavors,

$Q(\text{in}) * A(\text{in}) * Z(\text{in})^H = Q(\text{out}) * A(\text{out}) * Z(\text{out})^H$

$Q(\text{in}) * B(\text{in}) * Z(\text{in})^H = Q(\text{out}) * B(\text{out}) * Z(\text{out})^H$.

Input Parameters

wantq LOGICAL.

If *wantq* = .TRUE. : update the left transformation matrix *Q*;

If *wantq* = .FALSE. : do not update *Q*.

wantz — LOGICAL.

If *wantz* = .TRUE. : update the right transformation matrix *Z*;

If *wantz* = .FALSE.: do not update *Z*.

n — INTEGER. The order of the matrices *A* and *B*. *n*≥ 0.

a, *b* — REAL for stgex2DOUBLE PRECISION for dtgex2

COMPLEX for ctgex2

DOUBLE COMPLEX for ztgex2.

Arrays, DIMENSION (*lda*, *n*) and (*ldb*, *n*), respectively.

On entry, the matrices *A* and *B* in the pair (*A*, *B*).

lda — INTEGER. The leading dimension of the array *a*. *lda*≥ max(1,*n*).

ldb — INTEGER. The leading dimension of the array *b*. *ldb*≥ max(1,*n*).

q, *z* — REAL for stgex2DOUBLE PRECISION for dtgex2

COMPLEX for ctgex2

DOUBLE COMPLEX for ztgex2.

Arrays, DIMENSION (*ldq*, *n*) and (*ldz*, *n*), respectively.

On entry, if *wantq* = .TRUE., *q* contains the orthogonal/unitary matrix *Q*, and if *wantz* = .TRUE., *z* contains the orthogonal/unitary matrix *Z*.

ldq — INTEGER. The leading dimension of the array *q*. *ldq*≥ 1.

If *wantq* = .TRUE., *ldq*≥*n*.

ldz — INTEGER. The leading dimension of the array *z*. *ldz*≥ 1.

If *wantz* = .TRUE., *ldz*≥*n*.

j1 — INTEGER.

The index to the first block (*A*11, *B*11). 1 ≤ *j1* ≤ *n*.

n1 — INTEGER. Used with real flavors only. The order of the first block (*A*11, *B*11). *n1* = 0, 1 or 2.

n2 — INTEGER. Used with real flavors only. The order of the second block (*A*22, *B*22). *n2* = 0, 1 or 2.

work — REAL for stgex2

DOUBLE PRECISION for dtgex2.

Workspace array, DIMENSION(max(1,*lwork*)). Used with real flavors only.

lwork — INTEGER. The dimension of the array *work*.

lwork≥max(*n**(*n2*+*n1*), 2*(*n2*+*n1*)^2)

Output Parameters

a — On exit, the updated matrix *A*.

B	On exit, the updated matrix *B*.
Q	On exit, the updated matrix *Q*. Not referenced if *wantq* = .FALSE..
z	On exit, the updated matrix *Z*. Not referenced if *wantz* = .FALSE..
info	INTEGER. =0: Successful exit For stgex2/dtgex2: If *info* = 1, the transformed matrix (*A*, *B*) would be too far from generalized Schur form; the blocks are not swapped and (*A*, *B*) and (*Q*, *Z*) are unchanged. The problem of swapping is too ill-conditioned. If *info* = -16: *lwork* is too small. Appropriate value for *lwork* is returned in *work*(1). For ctgex2/ztgex2: If *info* = 1, the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is ill-conditioned.

?tgsy2

Solves the generalized Sylvester equation (unblocked algorithm).

Syntax

call stgsy2(*trans*, *ijob*, *m*, *n*, *a*, *lda*, *b*, *ldb*, *c*, *ldc*, *d*, *ldd*, *e*, *lde*, *f*, *ldf*, *scale*, *rdsum*, *rdscal*, *iwork*, *pq*, *info*)

call dtgsy2(*trans*, *ijob*, *m*, *n*, *a*, *lda*, *b*, *ldb*, *c*, *ldc*, *d*, *ldd*, *e*, *lde*, *f*, *ldf*, *scale*, *rdsum*, *rdscal*, *iwork*, *pq*, *info*)

call ctgsy2(*trans*, *ijob*, *m*, *n*, *a*, *lda*, *b*, *ldb*, *c*, *ldc*, *d*, *ldd*, *e*, *lde*, *f*, *ldf*, *scale*, *rdsum*, *rdscal*, *iwork*, *pq*, *info*)

call ztgsy2(*trans*, *ijob*, *m*, *n*, *a*, *lda*, *b*, *ldb*, *c*, *ldc*, *d*, *ldd*, *e*, *lde*, *f*, *ldf*, *scale*, *rdsum*, *rdscal*, *iwork*, *pq*, *info*)

Include Files

- mkl.fi

Description

The routine ?tgsy2 solves the generalized Sylvester equation:

*A*R-L*B=scale*C* (1)

*D*R-L*E=scale*F*

using Level 1 and 2 BLAS, where *R* and *L* are unknown *m*-by-*n* matrices, (*A*, *D*), (*B, E)* and (*C*, *F*) are given matrix pairs of size *m*-by -*m*, *n*-by-*n* and *m*-by-*n*, respectively. For stgsy2/dtgsy2, pairs (*A*, *D*) and (*B*, *E*) must be in generalized Schur canonical form, that is, *A*, *B* are upper quasi triangular and *D*, *E* are upper triangular. For ctgsy2/ztgsy2, matrices *A*, *B*, *D* and *E* are upper triangular (that is, (*A*, *D*) and (*B*, *E*) in generalized Schur form).

The solution (*R*, *L*) overwrites (*C*, *F*).

$0 \le scale \le 1$ is an output scaling factor chosen to avoid overflow.

In matrix notation, solving equation (1) corresponds to solve

`Z*x = scale*b`

where *Z* is defined for real flavors as

$$Z = \begin{bmatrix} \text{kron}(I_n, A) & -\text{kron}(B^T, I_m) \\ \text{kron}(I_n, D) & -\text{kron}(E^T, I_m) \end{bmatrix} \quad (2)$$

and for complex flavors as

$$Z = \begin{bmatrix} \text{kron}(I_n, A) & -\text{kron}(B^H, I_m) \\ \text{kron}(I_n, D) & -\text{kron}(E^H, I_m) \end{bmatrix} \quad (3)$$

Here I_k is the identity matrix of size *k* and X^T (X^H) is the transpose (conjugate transpose) of *X*. `kron`(*X*, *Y*) denotes the Kronecker product between the matrices *X* and *Y*.

For real flavors, if `trans = 'T'`, solve the transposed system

$Z^T*y = scale*b$

for *y*, which is equivalent to solving for *R* and *L* in

$$A^T*R+D^T*L=scale*C \quad (4)$$

$$R*B^T+L*E^T=scale*(-F)$$

For complex flavors, if `trans = 'C'`, solve the conjugate transposed system

$Z^H*y = scale*b$

for *y*, which is equivalent to solving for *R* and *L* in

$$A^H*R+D^H*L=scale*C \quad (5)$$

$$R*B^H+L*E^H=scale*(-F)$$

These cases are used to compute an estimate of `Dif[(A,D),(B,E)] = sigma_min(Z)` using reverse communication with `?lacon`.

`?tgsy2` also (for `ijob`≥ `1`) contributes to the computation in `?tgsyl` of an upper bound on the separation between two matrix pairs. Then the input (*A*, *D*), (*B*, *E*) are sub-pencils of the matrix pair (two matrix pairs) in `?tgsyl`. See `?tgsyl` for details.

Input Parameters

`trans`	`CHARACTER*1`. If `trans = 'N'`, solve the generalized Sylvester equation (1); If `trans = 'T'`: solve the transposed system (4). If `trans = 'C'`: solve the conjugate transposed system (5).
`ijob`	`INTEGER`. Specifies what kind of functionality is to be performed. If `ijob = 0`: solve (1) only.

If *ijob* = 1: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (look ahead strategy is used);

If *ijob* = 2: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (?gecon on sub-systems is used).

Not referenced if *trans* = 'T'.

m — INTEGER. On entry, *m* specifies the order of *A* and *D*, and the row dimension of *C*, *F*, *R* and *L*.

n — INTEGER. On entry, *n* specifies the order of *B* and *E*, and the column dimension of *C*, *F*, *R* and *L*.

a, *b* — REAL for stgsy2

DOUBLE PRECISION for dtgsy2

COMPLEX for ctgsy2

DOUBLE COMPLEX for ztgsy2.

Arrays, DIMENSION (*lda*, *m*) and (*ldb*, *n*), respectively. On entry, *a* contains an upper (quasi) triangular matrix *A*, and *b* contains an upper (quasi) triangular matrix *B*.

lda — INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1, *m*).

ldb — INTEGER.

The leading dimension of the array *b*. *ldb* ≥ max(1, *n*).

c, *f* — REAL for stgsy2

DOUBLE PRECISION for dtgsy2

COMPLEX for ctgsy2

DOUBLE COMPLEX for ztgsy2.

Arrays, DIMENSION (*ldc*, *n*) and (*ldf*, *n*), respectively. On entry, *c* contains the right-hand-side of the first matrix equation in (1), and *f* contains the right-hand-side of the second matrix equation in (1).

ldc — INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1, *m*).

d, *e* — REAL for stgsy2

DOUBLE PRECISION for dtgsy2

COMPLEX for ctgsy2

DOUBLE COMPLEX for ztgsy2.

Arrays, DIMENSION (*ldd*, *m*) and (*lde*, *n*), respectively. On entry, *d* contains an upper triangular matrix *D*, and *e* contains an upper triangular matrix *E*.

ldd — INTEGER. The leading dimension of the array *d*. *ldd* ≥ max(1, *m*).

lde — INTEGER. The leading dimension of the array *e*. *lde* ≥ max(1, *n*).

ldf — INTEGER. The leading dimension of the array *f*. *ldf* ≥ max(1, *m*).

rdsum

REAL for stgsy2/ctgsy2

DOUBLE PRECISION for dtgsy2/ztgsy2.

On entry, the sum of squares of computed contributions to the Dif-estimate under computation by ?tgsyL, where the scaling factor *rdscal* has been factored out.

rdscal

REAL for stgsy2/ctgsy2

DOUBLE PRECISION for dtgsy2/ztgsy2.

On entry, scaling factor used to prevent overflow in *rdsum*.

iwork

INTEGER. Used with real flavors only.

Workspace array, DIMENSION (*m*+*n*+2).

Output Parameters

c

On exit, if *ijob* = 0, *c* is overwritten by the solution *R*.

f

On exit, if *ijob* = 0, *f* is overwritten by the solution *L*.

scale

REAL for stgsy2/ctgsy2

DOUBLE PRECISION for dtgsy2/ztgsy2.

On exit, $0 \le scale \le 1$. If $0 < scale < 1$, the solutions *R* and *L* (*C* and *F* on entry) hold the solutions to a slightly perturbed system, but the input matrices *A*, *B*, *D* and *E* are not changed. If $scale = 0$, *R* and *L* hold the solutions to the homogeneous system with $C = F = 0$. Normally $scale = 1$.

rdsum

On exit, the corresponding sum of squares updated with the contributions from the current sub-system.

If *trans* = 'T', *rdsum* is not touched.

Note that *rdsum* only makes sense when ?tgsy2 is called by ?tgsyl.

rdscal

On exit, *rdscal* is updated with respect to the current contributions in *rdsum*.

If *trans* = 'T', *rdscal* is not touched.

Note that *rdscal* only makes sense when ?tgsy2 is called by ?tgsyl.

pq

INTEGER. Used with real flavors only.

On exit, the number of subsystems (of size 2-by-2, 4-by-4 and 8-by-8) solved by the routine stgsy2/dtgsy2.

info

INTEGER. On exit, if *info* is set to

= 0: Successful exit

< 0: If *info* = -*i*, the *i*-th argument has an illegal value.

> 0: The matrix pairs (*A*, *D*) and (*B*, *E*) have common or very close eigenvalues.

?trti2

Computes the inverse of a triangular matrix (unblocked algorithm).

Syntax

```
call strti2( uplo, diag, n, a, lda, info )
call dtrti2( uplo, diag, n, a, lda, info )
call ctrti2( uplo, diag, n, a, lda, info )
call ztrti2( uplo, diag, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine ?trti2 computes the inverse of a real/complex upper or lower triangular matrix.

This is the *Level 2 BLAS* version of the algorithm.

Input Parameters

uplo

CHARACTER*1.

Specifies whether the matrix *A* is upper or lower triangular.

= 'U': upper triangular

= 'L': lower triangular

diag

CHARACTER*1.

Specifies whether or not the matrix *A* is unit triangular.

= 'N': non-unit triangular

= 'N': non-unit triangular

n

INTEGER. The order of the matrix *A*. $n \geq 0$.

a

REAL for strti2

DOUBLE PRECISION for dtrti2

COMPLEX for ctrti2

DOUBLE COMPLEX for ztrti2.

Array, DIMENSION (*lda*, *n*).

On entry, the triangular matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced.

If *diag* = 'U', the diagonal elements of *a* are also not referenced and are assumed to be 1.

lda — INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a — On exit, the (triangular) inverse of the original matrix, in the same storage format.

info — INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument had an illegal value

clag2z

Converts a complex single precision matrix to a complex double precision matrix.

Syntax

```
call clag2z( m, n, sa, ldsa, a, lda, info )
```

Include Files

- mkl.fi

Description

This routine converts a complex single precision matrix *SA* to a complex double precision matrix *A*.

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is an auxiliary routine so there is no argument checking.

Input Parameters

m — INTEGER. The number of lines of the matrix *A* (*m* ≥ 0).

n — INTEGER. The number of columns in the matrix *A* (*n* ≥ 0).

ldsa — INTEGER. The leading dimension of the array *sa*; *ldsa* ≥ max(1, *m*).

a — DOUBLE PRECISION array, DIMENSION (*lda*, *n*).

On entry, contains the *m*-by-*n* coefficient matrix *A*.

lda — INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1, *m*).

Output Parameters

sa — REAL array, DIMENSION (*ldsa*, *n*).

On exit, contains the *m*-by-*n* coefficient matrix *SA*.

info — INTEGER.

If *info* = 0, the execution is successful.

dlag2s

Converts a double precision matrix to a single precision matrix.

Syntax

```
call dlag2s( m, n, a, lda, sa, ldsa, info )
```

Include Files

- mkl.fi

Description

This routine converts a double precision matrix *SA* to a single precision matrix *A*.

RMAX is the overflow for the single precision arithmetic. dlag2s checks that all the entries of *A* are between –RMAX and RMAX. If not, the convertion is aborted and a flag is raised.

This is an auxiliary routine so there is no argument checking.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in the matrix *A* (*n*≥ 0).
a	DOUBLE PRECISION array, DIMENSION (*lda*, *n*). On entry, contains the *m*-by-*n* coefficient matrix *A*.
lda	INTEGER. The leading dimension of the array *a*; *lda*≥ max(1, *m*).
ldsa	INTEGER. The leading dimension of the array *sa*; *ldsa*≥ max(1, *m*).

Output Parameters

sa	REAL array, DIMENSION (*ldsa*, *n*). On exit, if *info* = 0, contains the *m*-by-*n* coefficient matrix *SA*; if *info* > 0, the content of *sa* is unspecified.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = 1, an entry of the matrix *A* is greater than the single precision overflow threshold; in this case, the content of *sa* on exit is unspecified.

slag2d

Converts a single precision matrix to a double precision matrix.

Syntax

```
call slag2d( m, n, sa, ldsa, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine converts a single precision matrix *SA* to a double precision matrix *A*.

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is an auxiliary routine so there is no argument checking.

Input Parameters

m	INTEGER. The number of lines of the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in the matrix *A* (*n*≥ 0).
sa	REAL array, DIMENSION (*ldsa*, *n*). On entry, contains the *m*-by-*n* coefficient matrix *SA*.
ldsa	INTEGER. The leading dimension of the array *sa*; *ldsa*≥ max(1, *m*).
lda	INTEGER. The leading dimension of the array *a*; *lda*≥ max(1, *m*).

Output Parameters

a	DOUBLE PRECISION array, DIMENSION (*lda*, *n*). On exit, contains the *m*-by-*n* coefficient matrix *A*.
info	INTEGER. If *info* = 0, the execution is successful.

zlag2c

Converts a complex double precision matrix to a complex single precision matrix.

Syntax

```
call zlag2c( m, n, a, lda, sa, ldsa, info )
```

Include Files

- mkl.fi

Description

The routine converts a double precision complex matrix *SA* to a single precision complex matrix *A*.

RMAX is the overflow for the single precision arithmetic. zlag2c checks that all the entries of *A* are between –RMAX and RMAX. If not, the convertion is aborted and a flag is raised.

This is an auxiliary routine so there is no argument checking.

Input Parameters

m	INTEGER. The number of lines of the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns in the matrix *A* (*n*≥ 0).
a	DOUBLE COMPLEX array, DIMENSION (*lda*, *n*).

	On entry, contains the *m*-by-*n* coefficient matrix *A*.
lda	INTEGER. The leading dimension of the array *a*; *lda*≥ max(1, *m*).
ldsa	INTEGER. The leading dimension of the array *sa*; *ldsa*≥ max(1, *m*).

Output Parameters

sa	COMPLEX array, DIMENSION (*ldsa*, *n*). On exit, if *info* = 0, contains the *m*-by-*n* coefficient matrix *SA*; if *info* > 0, the content of *sa* is unspecified.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = 1, an entry of the matrix *A* is greater than the single precision overflow threshold; in this case, the content of *sa* on exit is unspecified.

?larfp

Generates a real or complex elementary reflector.

Syntax

```
call slarfp(n, alpha, x, incx, tau)
call dlarfp(n, alpha, x, incx, tau)
call clarfp(n, alpha, x, incx, tau)
call zlarfp(n, alpha, x, incx, tau)
```

Include Files

- mkl.fi

Description

The ?larfp routines generate a real or complex elementary reflector *H* of order *n*, such that

```
H * (alpha) = (beta),
    (  x  )   ( 0  )
```

and *H*'**H* =*I* for real flavors, conjg(*H*)'**H* =*I* for complex flavors.

Here

alpha and *beta* are scalars, *beta* is real and non-negative,

x is (*n*-1)-element vector.

H is represented in the form

```
H = I - tau*( 1 )* (1   v'),
            ( v )
```

where *tau* is scalar, and *v* is (*n*-1)-element vector .

For real flavors if the elements of *x* are all zero, then *tau* = 0 and *H* is taken to be the unit matrix. Otherwise 1 ≤*tau*≤ 2.

For complex flavors if the elements of *x* are all zero and *alpha* is real, then *tau* = 0 and *H* is taken to be the unit matrix. Otherwise 1 ≤ real(*tau*) ≤ 2, and abs (*tau*-1≤ 1.

Input Parameters

n	INTEGER. Specifies the order of the elementary reflector.
alpha	REAL for slarfp DOUBLE PRECISION for dlarfp COMPLEX for clarfp DOUBLE COMPLEX for zlarfp Specifies the scalar *alpha*.
x	REAL for slarfp DOUBLE PRECISION for dlarfp COMPLEX for clarfp DOUBLE COMPLEX for zlarfp Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). It contains the vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

alpha	Overwritten by the value *beta*.
y	Overwritten by the vector *v*.
tau	REAL for slarfp DOUBLE PRECISION for dlarfp COMPLEX for clarfp DOUBLE COMPLEX for zlarfp Contains the scalar *tau*.

ila?lc

Scans a matrix for its last non-zero column.

Syntax

value = ilaslc(*m*, *n*, *a*, *lda*)

value = iladlc(*m*, *n*, *a*, *lda*)

value = ilaclc(*m*, *n*, *a*, *lda*)

value = ilazlc(*m*, *n*, *a*, *lda*)

Include Files

- mkl.fi

Description

The ila?lc routines scan a matrix *A* for its last non-zero column.

Input Parameters

m	INTEGER. Specifies number of rows in the matrix *A*.
n	INTEGER. Specifies number of columns in the matrix *A*.
a	REAL for ilaslc DOUBLE PRECISION for iladlc COMPLEX for ilaclc DOUBLE COMPLEX for ilazlc Array, DIMENSION(*lda*, *). The second dimension of *a* must be at least max(1, *n*). Before entry the leading *n*-by-*n* part of the array *a* must contain the matrix *A*.
lda	INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).

Output Parameters

value	INTEGER Number of the last non-zero column.

ila?lr

Scans a matrix for its last non-zero row.

Syntax

```
value = ilaslr(m, n, a, lda)
value = iladlr(m, n, a, lda)
value = ilaclr(m, n, a, lda)
value = ilazlr(m, n, a, lda)
```

Include Files

- mkl.fi

Description

The ila?lr routines scan a matrix *A* for its last non-zero row.

Input Parameters

m	INTEGER. Specifies number of rows in the matrix *A*.
n	INTEGER. Specifies number of columns in the matrix *A*.

a	REAL for ilaslr DOUBLE PRECISION for iladlr COMPLEX for ilaclr DOUBLE COMPLEX for idazlr Array, DIMENSION(*lda*, *). The second dimension of *a* must be at least max(1, *n*). Before entry the leading *n*-by-*n* part of the array *a* must contain the matrix *A*.
lda	INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).

Output Parameters

value	INTEGER Number of the last non-zero row.

?gsvj0

Pre-processor for the routine ?gesvj.

Syntax

call sgsvj0(*jobv*, *m*, *n*, *a*, *lda*, *d*, *sva*, *mv*, *v*, *ldv*, *eps*, *sfmin*, *tol*, *nsweep*, *work*, *lwork*, *info*)

call dgsvj0(*jobv*, *m*, *n*, *a*, *lda*, *d*, *sva*, *mv*, *v*, *ldv*, *eps*, *sfmin*, *tol*, *nsweep*, *work*, *lwork*, *info*)

call cgsvj0(*jobv*, *m*, *n*, *a*, *lda*, *d*, *sva*, *mv*, *v*, *ldv*, *eps*, *sfmin*, *tol*, *nsweep*, *work*, *lwork*, *info*)

call zgsvj0(*jobv*, *m*, *n*, *a*, *lda*, *d*, *sva*, *mv*, *v*, *ldv*, *eps*, *sfmin*, *tol*, *nsweep*, *work*, *lwork*, *info*)

Include Files

- mkl.fi

Description

This routine is called from ?gesvj as a pre-processor and that is its main purpose. It applies Jacobi rotations in the same way as ?gesvj does, but it does not check convergence (stopping criterion).

The routine ?gsvj0 enables ?gesvj to use a simplified version of itself to work on a submatrix of the original matrix.

Input Parameters

jobv	CHARACTER*1. Must be 'V', 'A', or 'N'. Specifies whether the output from this routine is used to compute the matrix *V*.

If *jobv* = 'V', the product of the Jacobi rotations is accumulated by post-multiplying the *n*-by-*n* array *v*.

If *jobv* = 'A', the product of the Jacobi rotations is accumulated by post-multiplying the *mv*-by-*n* array *v*.

If *jobv* = 'N', the Jacobi rotations are not accumulated.

m INTEGER. The number of rows of the input matrix *A* ($m \geq 0$).

n INTEGER. The number of columns of the input matrix *B* ($m \geq n \geq 0$).

a REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

COMPLEX for cgsvj0

DOUBLE COMPLEX for zgsvj0

Array, DIMENSION (*lda*, *n*). Contains the *m*-by-*n* matrix *A*, such that *A**diag(*D*) represents the input matrix.

lda INTEGER. The leading dimension of *a*; at least max(1, *m*).

d REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

COMPLEX for cgsvj0

DOUBLE COMPLEX for zgsvj0

Array, DIMENSION (*n*). Contains the diagonal matrix *D* that accumulates the scaling factors from the fast scaled Jacobi rotations. On entry *A**diag(*D*) represents the input matrix.

sva REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

REAL for cgsvj0

DOUBLE PRECISION for zgsvj0.

Array, DIMENSION (*n*). Contains the Euclidean norms of the columns of the matrix *A**diag(*D*).

mv INTEGER. The leading dimension of *b*; at least max(1, *p*).

If *jobv* = 'A', then *mv* rows of *v* are post-multiplied by a sequence of Jacobi rotations.

If *jobv* = 'N', then *mv* is not referenced .

v REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

COMPLEX for cgsvj0

DOUBLE COMPLEX for zgsvj0

Array, DIMENSION (*ldv*, *n*).

If *jobv* = 'V', then *n* rows of *v* are post-multiplied by a sequence of Jacobi rotations.

If *jobv* = 'A', then *mv* rows of *v* are post-multiplied by a sequence of Jacobi rotations.

If *jobv* = 'N', then *v* is not referenced.

ldv

INTEGER. The leading dimension of the array *v*; *ldv* ≥ 1

ldv ≥ *n* if *jobv* = 'V';

ldv ≥ *mv* if *jobv* = 'A'.

eps

REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

REAL for cgsvj0

DOUBLE PRECISION for zgsvj0.

The relative machine precision (epsilon) returned by the routine ?lamch.

sfmin

REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

REAL for cgsvj0

DOUBLE PRECISION for zgsvj0.

Value of safe minimum returned by the routine ?lamch.

tol

REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

REAL for cgsvj0

DOUBLE PRECISION for zgsvj0.

The threshold for Jacobi rotations. For a pair *A*(:,p), *A*(:,q) of pivot columns, the Jacobi rotation is applied only if abs(cos(angle(*A*(:,p),*A*(:,q)))) > *tol*.

nsweep

INTEGER.

The number of sweeps of Jacobi rotations to be performed.

work

REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

COMPLEX for cgsvj0

DOUBLE COMPLEX for zgsvj0

Workspace array, DIMENSION (*lwork*).

lwork

INTEGER. The size of the array *work*; at least max(1, *m*).

Output Parameters

a	On exit, *A**diag(*D*) represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in *tol* and *nsweep*, respectively
d	On exit, *A**diag(*D*) represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in *tol* and *nsweep*, respectively.
sva	On exit, contains the Euclidean norms of the columns of the output matrix *A**diag(*D*).
v	If *jobv* = 'V', then *n* rows of *v* are post-multiplied by a sequence of Jacobi rotations. If *jobv* = 'A', then *mv* rows of *v* are post-multiplied by a sequence of Jacobi rotations. If *jobv* = 'N', then *v* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

?gsvj1

Pre-processor for the routine ?gesvj*, applies Jacobi rotations targeting only particular pivots.*

Syntax

```
call sgsvj1(jobv, m, n, n1, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep, work, lwork, info)
call dgsvj1(jobv, m, n, n1, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep, work, lwork, info)
call cgsvj1(jobv, m, n, n1, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep, work, lwork, info)
call zgsvj1(jobv, m, n, n1, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep, work, lwork, info)
```

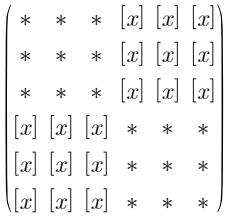
Include Files

- mkl.fi

Description

This routine is called from ?gesvj as a pre-processor and that is its main purpose. It applies Jacobi rotations in the same way as ?gesvj does, but it targets only particular pivots and it does not check convergence (stopping criterion).

The routine ?gsvj1 applies few sweeps of Jacobi rotations in the column space of the input *m*-by-*n* matrix *A*. The pivot pairs are taken from the (1,2) off-diagonal block in the corresponding *n*-by-*n* Gram matrix *A*'**A*. The block-entries (*tiles*) of the (1,2) off-diagonal block are marked by the [x]'s in the following scheme:



row-cycling in the *nblr*-by-*nblc*[x] blocks, row-cyclic pivoting inside each [x] block

In terms of the columns of the matrix *A*, the first *n1* columns are rotated 'against' the remaining *n*-*n1* columns, trying to increase the angle between the corresponding subspaces. The off-diagonal block is *n1*-by-(*n*-*n1*) and it is tiled using quadratic tiles. The number of sweeps is specified by *nsweep*, and the orthogonality threshold is set by *tol*.

Input Parameters

jobv

CHARACTER*1. Must be 'V', 'A', or 'N'.

Specifies whether the output from this routine is used to compute the matrix *V*.

If *jobv* = 'V', the product of the Jacobi rotations is accumulated by post-multiplying the *n*-by-*n* array *v*.

If *jobv* = 'A', the product of the Jacobi rotations is accumulated by post-multiplying the *mv*-by-*n* array *v*.

If *jobv* = 'N', the Jacobi rotations are not accumulated.

m

INTEGER. The number of rows of the input matrix *A* ($m \geq 0$).

n

INTEGER. The number of columns of the input matrix *B* ($m \geq n \geq 0$).

n1

INTEGER. Specifies the 2-by-2 block partition. The first *n1* columns are rotated 'against' the remaining *n*–*n1* columns of the matrix *A*.

a

REAL for sgsvj1

DOUBLE PRECISION for dgsvj1.

COMPLEX for cgsvj1

DOUBLE COMPLEX for zgsvj1.

Array, DIMENSION (*lda*, *n*). Contains the *m*-by-*n* matrix *A*, such that *A**diag(*D*) represents the input matrix.

lda

INTEGER. The leading dimension of *a*; at least max(1, *m*).

d

REAL for sgsvj1

DOUBLE PRECISION for dgsvj1.

COMPLEX for cgsvj1

DOUBLE COMPLEX for zgsvj1.

Arrays, DIMENSION (*n*). Contains the diagonal matrix *D* that accumulates the scaling factors from the fast scaled Jacobi rotations. On entry *A**diag(*D*) represents the input matrix.

sva

REAL for sgsvj1

DOUBLE PRECISION for dgsvj1.

REAL for cgsvj1

DOUBLE PRECISION for zgsvj1.

Arrays, DIMENSION (*n*). Contains the Euclidean norms of the columns of the matrix *A**diag(*D*).

mv INTEGER. The leading dimension of *b*; at least max(1, *p*).

If *jobv* = 'A', then *mv* rows of *v* are post-multiplied by a sequence of Jacobi rotations.

If *jobv* = 'N', then *mv* is not referenced .

v REAL for sgsvj1

DOUBLE PRECISION for dgsvj1.

COMPLEX for cgsvj1

DOUBLE COMPLEX for zgsvj1.

Array, DIMENSION (*ldv*, *n*).

If *jobv* = 'V', then *n* rows of *v* are post-multiplied by a sequence of Jacobi rotations.

If *jobv* = 'A', then *mv* rows of *v* are post-multiplied by a sequence of Jacobi rotations.

If *jobv* = 'N', then *v* is not referenced.

ldv INTEGER. The leading dimension of the array *v*; *ldv*≥ 1

ldv≥ *n* if *jobv* = 'V';

ldv≥ *mv* if *jobv* = 'A'.

eps REAL for sgsvj1

DOUBLE PRECISION for dgsvj1.

REAL for cgsvj1

DOUBLE PRECISION for zgsvj1.

The relative machine precision (epsilon) returned by the routine ?lamch.

sfmin REAL for sgsvj1

DOUBLE PRECISION for dgsvj1.

REAL for cgsvj1

DOUBLE PRECISION for zgsvj1.

Value of safe minimum returned by the routine ?lamch.

tol REAL for sgsvj1

DOUBLE PRECISION for dgsvj1.

REAL for cgsvj1

DOUBLE PRECISION for zgsvj1.

	The threshold for Jacobi rotations. For a pair `A(:,p)`, `A(:,q)` of pivot columns, the Jacobi rotation is applied only if `abs(cos(angle(A(:,p),A(:,q))))> tol`.
nsweep	`INTEGER`. The number of sweeps of Jacobi rotations to be performed.

work	`REAL` for `sgsvj1`
	`DOUBLE PRECISION` for `dgsvj1`.
	`COMPLEX` for `cgsvj1`
	`DOUBLE COMPLEX` for `zgsvj1`.
	Workspace array, `DIMENSION` (*lwork*).

lwork	`INTEGER`. The size of the array *work*; at least `max(1, m)`.

Output Parameters

a	On exit, `A*diag(D)` represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in *tol* and *nsweep*, respectively
d	On exit, `A*diag(D)` represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in *tol* and *nsweep*, respectively.
sva	On exit, contains the Euclidean norms of the columns of the output matrix `A*diag(D)`.
v	If *jobv* = `'V'`, then *n* rows of *v* are post-multiplied by a sequence of Jacobi rotations. If *jobv* = `'A'`, then *mv* rows of *v* are post-multiplied by a sequence of Jacobi rotations. If *jobv* = `'N'`, then *v* is not referenced.
info	`INTEGER`. If *info* = `0`, the execution is successful. If *info* = `-i`, the *i*-th parameter had an illegal value.

?sfrk

Performs a symmetric rank-k operation for matrix in RFP format.

Syntax

```
call ssfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
call dsfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
```

Include Files

- `mkl.fi`

Description

The ?sfrk routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

$C := alpha*A*A^T + beta*C$,

or

$C := alpha*A^T*A + beta*C$,

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix in rectangular full packed (RFP) format,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

transr

CHARACTER*1.

if *transr* = 'N' or 'n', the normal form of RFP *C* is stored;

if *transr*= 'T' or 't', the transpose form of RFP *C* is stored.

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is used.

If *uplo* = 'U' or 'u', then the upper triangular part of the array *c* is used.

If *uplo* = 'L' or 'l', then the low triangular part of the array *c* is used.

trans

CHARACTER*1. Specifies the operation:

if *trans* = 'N' or 'n', then $C := alpha*A*A^T + beta*C$;

if *trans* = 'T' or 't', then $C := alpha*A^T*A + beta*C$;

n

INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

k

INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *A*, and on entry with *trans* = 'T' or 't', *k* specifies the number of rows of the matrix *A*.

The value of *k* must be at least zero.

alpha

REAL for ssfrk

DOUBLE PRECISION for dsfrk

Specifies the scalar *alpha*.

a

REAL for ssfrk

DOUBLE PRECISION for dsfrk

Array, DIMENSION(*lda*,*ka*), where *ka* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda — INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *lda* must be at least max(1,*n*), otherwise *lda* must be at least max(1, *k*).

beta — REAL for ssfrk

DOUBLE PRECISION for dsfrk

Specifies the scalar *beta*.

c — REAL for ssfrk

DOUBLE PRECISION for dsfrk

Array, size (*n**(*n*+1)/2). Before entry contains the symmetric matrix *C* in RFP format.

Output Parameters

c — If *trans* = 'N' or 'n', then *c* contains *C* := *alpha***A***A*' + *beta***C*;

if *trans* = 'T' or 't', then *c* contains *C* := *alpha***A*'**A* + *beta***C*;

?hfrk

Performs a Hermitian rank-k operation for matrix in RFP format.

Syntax

```
call chfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
call zhfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
```

Include Files

- mkl.fi

Description

The ?hfrk routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

$C := alpha*A*A^H + beta*C,$

or

$C := alpha*A^H*A + beta*C,$

where:

alpha and *beta* are real scalars,

C is an *n*-by-*n* Hermitian matrix in RFP format,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

transr — CHARACTER*1.

if *transr* = 'N' or 'n', the normal form of RFP *C* is stored;

if *transr* = 'C' or 'c', the conjugate-transpose form of RFP *C* is stored.

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is used.

If *uplo* = 'U' or 'u', then the upper triangular part of the array *c* is used.

If *uplo* = 'L' or 'l', then the low triangular part of the array *c* is used.

trans

CHARACTER*1. Specifies the operation:

if *trans* = 'N' or 'n', then $C := alpha*A*A^H + beta*C$;

if *trans* = 'C' or 'c', then $C := alpha*A^H*A + beta*C$.

n

INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

k

INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *a*, and on entry with *trans* = 'T' or 't' or 'C' or 'c', *k* specifies the number of rows of the matrix *a*.

The value of *k* must be at least zero.

alpha

COMPLEX for chfrk

DOUBLE COMPLEX for zhfrk

Specifies the scalar *alpha*.

a

COMPLEX for chfrk

DOUBLE COMPLEX for zhfrk

Array, DIMENSION(*lda*,*ka*), where *ka* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *lda* must be at least max(1,*n*), otherwise *lda* must be at least max(1, *k*).

beta

COMPLEX for chfrk

DOUBLE COMPLEX for zhfrk

Specifies the scalar *beta*.

c

COMPLEX for chfrk

DOUBLE COMPLEX for zhfrk

Array, size (*n**(*n*+1)/2). Before entry contains the Hermitian matrix *C* in in RFP format.

Output Parameters

c

If *trans* = 'N' or 'n', then *c* contains $C := alpha*A*A^H + beta*C$;

if *trans* = 'C' or 'c', then *c* contains $C := alpha*A^H*A + beta*C$;

?tfsm

Solves a matrix equation (one operand is a triangular matrix in RFP format).

Syntax

```
call stfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
call dtfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
call ctfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
call ztfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
```

Include Files

- mkl.fi

Description

The ?tfsm routines solve one of the following matrix equations:

```
op(A)*X = alpha*B,
```

or

```
X*op(A) = alpha*B,
```

where:

alpha is a scalar,

X and *B* are *m*-by-*n* matrices,

A is a unit, or non-unit, upper or lower triangular matrix in rectangular full packed (RFP) format.

op(*A*) can be one of the following:

- op(*A*) = *A* or op(*A*) = A^T for real flavors
- op(*A*) = *A* or op(*A*) = A^H for complex flavors

The matrix *B* is overwritten by the solution matrix *X*.

Input Parameters

transr

CHARACTER*1.

if *transr* = 'N' or 'n', the normal form of RFP *A* is stored;

if *transr* = 'T' or 't', the transpose form of RFP *A* is stored;

if *transr* = 'C' or 'c', the conjugate-transpose form of RFP *A* is stored.

side

CHARACTER*1. Specifies whether op(*A*) appears on the left or right of *X* in the equation:

if *side* = 'L' or 'l', then op(*A*)**X* = *alpha***B*;

if *side* = 'R' or 'r', then *X**op(*A*) = *alpha***B*.

uplo

CHARACTER*1. Specifies whether the RFP matrix *A* is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans — CHARACTER*1. Specifies the form of op(*A*) used in the matrix multiplication:

if *trans* = 'N' or 'n', then op(*A*) = *A*;

if *trans* = 'T' or 't', then op(*A*) = *A*';

if *trans* = 'C' or 'c', then op(*A*) = conjg(*A*').

diag — CHARACTER*1. Specifies whether the RFP matrix *A* is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

m — INTEGER. Specifies the number of rows of *B*. The value of *m* must be at least zero.

n — INTEGER. Specifies the number of columns of *B*. The value of *n* must be at least zero.

alpha — REAL for stfsm

DOUBLE PRECISION for dtfsm

COMPLEX for ctfsm

DOUBLE COMPLEX for ztfsm

Specifies the scalar *alpha*.

When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.

a — REAL for stfsm

DOUBLE PRECISION for dtfsm

COMPLEX for ctfsm

DOUBLE COMPLEX for ztfsm

Array, size (*n**(*n*+1)/2). Contains the matrix *A* in RFP format.

b — REAL for stfsm

DOUBLE PRECISION for dtfsm

COMPLEX for ctfsm

DOUBLE COMPLEX for ztfsm

Array, size (1, *ldb***n*)

Before entry, the leading *m*-by-*n* part of the array *b* must contain the right-hand side matrix *B*.

ldb — INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least max(1, *m*).

Output Parameters

b — Overwritten by the solution matrix *X*.

?lansf

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix in RFP format.

Syntax

```
val = slansf(norm, transr, uplo, n, a, work)
val = dlansf(norm, transr, uplo, n, a, work)
```

Include Files

- mkl.fi

Description

T

The function ?lansf returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* real symmetric matrix *A* in the rectangular full packed (RFP) format .

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

transr

CHARACTER*1.

Specifies whether the RFP format of matrix *A* is normal or transposed format.

If *transr* = 'N': RFP format is normal;

if *transr* = 'T': RFP format is transposed.

uplo

CHARACTER*1.

Specifies whether the RFP matrix *A* came from upper or lower triangular matrix.

If *uplo* = 'U': RFP matrix *A* came from an upper triangular matrix;

if *uplo* = 'L': RFP matrix *A* came from a lower triangular matrix.

n

INTEGER. The order of the matrix *A*. $n \geq 0$.

When *n* = 0, ?lansf is set to zero.

a

REAL for slansf

DOUBLE PRECISION for dlansf

Array, DIMENSION (*n**(*n*+1)/2).

The upper (if *uplo* = 'U') or lower (if *uplo* = 'L') part of the symetric matrix *A* stored in RFP format.

work

REAL for slansf.

DOUBLE PRECISION for dlansf.

Workspace array, DIMENSION(max(1,*lwork*)), where

lwork≥*n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val

REAL for slansf

DOUBLE PRECISION for dlansf

Value returned by the function.

?lanhf

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian matrix in RFP format.

Syntax

val = clanhf(*norm*, *transr*, *uplo*, *n*, *a*, *work*)

val = zlanhf(*norm*, *transr*, *uplo*, *n*, *a*, *work*)

Include Files

- mkl.fi

Description

The function ?lanhf returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* complex Hermitian matrix *A* in the rectangular full packed (RFP) format.

Input Parameters

norm

CHARACTER*1.

Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

transr CHARACTER*1.

Specifies whether the RFP format of matrix *A* is normal or conjugate-transposed format.

If *transr* = 'N': RFP format is normal;

if *transr* = 'C': RFP format is conjugate-transposed.

uplo CHARACTER*1.

Specifies whether the RFP matrix *A* came from upper or lower triangular matrix.

If *uplo* = 'U': RFP matrix *A* came from an upper triangular matrix;

if *uplo* = 'L': RFP matrix *A* came from a lower triangular matrix.

n INTEGER. The order of the matrix *A*. $n \geq 0$.

When *n* = 0, ?lanhf is set to zero.

a COMPLEX for clanhf

DOUBLE COMPLEX for zlanhf

Array, DIMENSION ($n*(n+1)/2$).

The upper (if *uplo* = 'U') or lower (if *uplo* = 'L') part of the Hermitian matrix *A* stored in RFP format.

work COMPLEX for clanhf.

DOUBLE COMPLEX for zlanhf.

Workspace array, DIMENSION(max(1,*lwork*)), where

lwork≥*n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val COMPLEX for clanhf

DOUBLE COMPLEX for zlanhf

Value returned by the function.

?tfttp

Copies a triangular matrix from the rectangular full packed format (TF) to the standard packed format (TP) .

Syntax

```
call stfttp( transr, uplo, n, arf, ap, info )
call dtfttp( transr, uplo, n, arf, ap, info )
call ctfttp( transr, uplo, n, arf, ap, info )
call ztfttp( transr, uplo, n, arf, ap, info )
```

Include Files

- mkl.fi

Description

The routine copies a triangular matrix *A* from the Rectangular Full Packed (RFP) format to the standard packed format. For the description of the RFP format, see Matrix Storage Schemes.

Input Parameters

transr CHARACTER*1.

= 'N': *arf* is in the Normal format,

= 'T': *arf* is in the Transpose format (for stfttp and dtfttp),

= 'C': *arf* is in the Conjugate-transpose format (for ctfttp and ztfttp).

uplo CHARACTER*1.

Specifies whether *A* is upper or lower triangular:

= 'U': A is upper triangular,

= 'L': A is lower triangular.

n INTEGER. The order of the matrix *A*. $n \geq 0$.

arf REAL for stfttp,

DOUBLE PRECISION for dtfttp,

COMPLEX for ctfttp,

DOUBLE COMPLEX for ztfttp.

Array, size at least *max* (1, $n*(n+1)/2$).

On entry, the upper or lower triangular matrix *A* stored in the RFP format.

Output Parameters

ap REAL for stfttp,

DOUBLE PRECISION for dtfttp,

COMPLEX for ctfttp,

DOUBLE COMPLEX for ztfttp.

Array, size at least *max* (1, $n*(n+1)/2$).

On exit, the upper or lower triangular matrix *A*, packed columnwise in a linear array.

The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', $ap(i + (j-1)*j/2) = A(i,j)$ for $1 \leq i \leq j$,

if *uplo* = 'L', $ap(i + (j-1)*(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

info INTEGER. If $info = 0$, the execution is successful.

If $info < 0$, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?tfttr

Copies a triangular matrix from the rectangular full packed format (TF) to the standard full format (TR) .

Syntax

```
call stfttr( transr, uplo, n, arf, a, lda, info )
call dtfttr( transr, uplo, n, arf, a, lda, info )
call ctfttr( transr, uplo, n, arf, a, lda, info )
call ztfttr( transr, uplo, n, arf, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine copies a triangular matrix *A* from the Rectangular Full Packed (RFP) format to the standard full format. For the description of the RFP format, see Matrix Storage Schemes.

Input Parameters

transr — CHARACTER*1.

= 'N': *arf* is in the Normal format,

= 'T': *arf* is in the Transpose format (for stfttr and dtfttr),

= 'C': *arf* is in the Conjugate-transpose format (for ctfttr and ztfttr).

uplo — CHARACTER*1.

Specifies whether *A* is upper or lower triangular:

= 'U': A is upper triangular,

= 'L': A is lower triangular.

n — INTEGER. The order of the matrices *arf* and *a*. $n \geq 0$.

arf — REAL for stfttr,

DOUBLE PRECISION for dtfttr,

COMPLEX for ctfttr,

DOUBLE COMPLEX for ztfttr.

Array, size at least *max* (1, $n*(n+1)/2$).

On entry, the upper or lower triangular matrix *A* stored in the RFP format.

lda — INTEGER. The leading dimension of the array *a*. $lda \geq max(1,n)$.

Output Parameters

a	REAL for stfttr, DOUBLE PRECISION for dtfttr, COMPLEX for ctfttr, DOUBLE COMPLEX for ztfttr. Array, size (*lda, **). On exit, the triangular matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* < 0, the *i*-th parameter had an illegal value. If *info* = –1011, memory allocation error occurred.

?tplqt2

?tplqt2 *computes an LQ factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal using the compact WY representation for Q.*

```
call stplqt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call dtplqt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call ctplqt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call ztplqt2(m, n, l, a, lda, b, ldb, t, ldt, info)
```

Description

?tplqt2 computes a LQ a factorization of a real or complex "triangular-pentagonal" matrix *C*, which is composed of a triangular block *A* and pentagonal block *B*, using the compact WY representation for *Q*.

The input matrix *C* is an *m*-by-(*m*+*n*) matrix:

C = [*A*] [*B*]

where *A* is a lower triangular *m*-by-*m* matrix, and *B* is an *m*-by-*n* pentagonal matrix consisting of an *m*-by-(*n*-*l*) rectangular matrix *B1* to the left of an *m*-by-*l* lower trapezoidal matrix *B2*:

[*B*] = [*B1*] [*B2*]

[*B1*] <- *m*-by-(*n*-*l*) rectangular

[*B2*] <- *m*-by-*l* lower trapezoidal.

The lower trapezoidal matrix *B2* consists of the first *l* columns of an *n*-by-*n* lower triangular matrix, where 0 ≤*l*≤ min(*m*,*n*). If *l*=0, *b* is rectangular *m*-by-*n*; if *m*=*l*=*n*, *b* is lower triangular.

The matrix *W* stores the elementary reflectors *H*(*i*) in the *i*-th row above the diagonal (of *A*) in the *m*-by-(*m* +*n*) input matrix *C*:

[*C*] = [*A*] [*B*]

[*A*] <- lower triangular *m*-by-*m*

[*B*] <- *m*-by-*n* pentagonal

so that *W* can be represented as

[*W*] = [*I*] [*V*]

[*I*] <- *m*-by-*m* identity matrix

[*V*] <- *m*-by-*n*, same form as *B*.

Thus, all of information needed for *W* is contained on exit in the array *b*, called *V* in the preceding. Note that *V* has the same form as *B*; that is,

[*V*] = [*V1*] [*V2*]

[*V1*] <- *m*-by-(*n*-*l*) rectangular

[*V2*] <- *m*-by-*l* lower trapezoidal.

The rows of *V* represent the vectors which define the *H*(*i*) elementary reflectors .

The (*m*+*n*)-by-(*m*+*n*) block reflector *H* is then given by $H = I - W^H * T * W$ where W^H is the conjugate transpose of *W* and *T* is the upper triangular factor of the block reflector.

Input Parameters

m	INTEGER. The total number of rows of the matrix *B*. *m*≥ 0.
n	INTEGER. The number of columns of the matrix *B*, and the order of the triangular matrix *A*. *n*≥ 0.
l	INTEGER. The number of rows of the lower trapezoidal part of *B*. min(*m*, *n*) ≥*l*≥ 0.
a	REAL for stplqt2 DOUBLE PRECISION for dtplqt2 COMPLEX for ctplqt2 COMPLEX*16 for ztplqt2 Array of size (*lda*,*m*). On entry, the lower triangular *m*-by-*m* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda*≥ max(1, *m*).
b	REAL for stplqt2 DOUBLE PRECISION for dtplqt2 COMPLEX for ctplqt2 COMPLEX*16 for ztplqt2 Array of size (*ldb*,*n*). On entry, the pentagonal *m*-by-*n* matrix *B*. The first *n*-*l* columns are rectangular, and the last *l* columns are lower trapezoidal.
ldb	INTEGER. The leading dimension of the array *b*. *ldb*≥ max(1, *m*).
ldt	INTEGER. The leading dimension of the array *t*. *ldt*≥ max(1, *m*)

Output Parameters

a	On exit, the elements on and below the diagonal of the array contain the lower triangular matrix *L*.

b	On exit, *b* contains the pentagonal matrix *V*.
t	REAL for stplqt2 DOUBLE PRECISION for dtplqt2 COMPLEX for ctplqt2 COMPLEX*16 for ztplqt2 Array of size (*ldt*,*m*). The *n*-by-*n* upper triangular factor *T* of the block reflector.
info	INTEGER. *info* = 0: successful exit. *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?tpqrt2

Computes a QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q.

Syntax

```
call stpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call dtpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call ctpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call ztpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call tpqrt2(a, b, t [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The input matrix *C* is an (*n*+*m*)-by-*n* matrix

$$C = \begin{bmatrix} A \\ B \end{bmatrix} \begin{matrix} \leftarrow n \times n \text{ upper triangular} \\ \leftarrow m \times n \text{ pentagonal} \end{matrix}$$

where *A* is an *n*-by-*n* upper triangular matrix, and *B* is an *m*-by-*n* pentagonal matrix consisting of an (*m*-*l*)-by-*n* rectangular matrix *B1* on top of an *l*-by-*n* upper trapezoidal matrix *B2*:

$$B = \begin{bmatrix} B1 \\ B2 \end{bmatrix} \begin{matrix} \leftarrow (m-l)\times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{matrix}$$

The upper trapezoidal matrix *B2* consists of the first l rows of an n-by-n upper triangular matrix, where $0 \le l \le \min(m,n)$. If $l=0$, *B* is an m-by-n rectangular matrix. If $m=l=n$, *B* is upper triangular. The matrix *W* contains the elementary reflectors *H(i)* in the *i*th column below the diagonal (of *A*) in the $(n+m)$-by-n input matrix *C* so that *W* can be represented as

$$W = \begin{bmatrix} I \\ V \end{bmatrix} \begin{matrix} \leftarrow n \times n \text{ identity} \\ \leftarrow m \times n \text{ pentagonal} \end{matrix}$$

Thus, *V* contains all of the information needed for *W*, and is returned in array b.

NOTE

V has the same form as *B*:

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix} \begin{matrix} \leftarrow (m-l)\times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{matrix}$$

The columns of *V* represent the vectors which define the *H(i)*s.

The $(m+n)$-by-$(m+n)$ block reflector *H* is then given by

$H = I - W*T*W^T$ for real flavors, and

$H = I - W*T*W^H$ for complex flavors

where W^T is the transpose of *W*, W^H is the conjugate transpose of *W*, and *T* is the upper triangular factor of the block reflector.

Input Parameters

m	INTEGER. The total number of rows in the matrix *B* ($m \ge 0$).
n	INTEGER. The number of columns in *B* and the order of the triangular matrix *A* ($n \ge 0$).
l	INTEGER. The number of rows of the upper trapezoidal part of *B* ($\min(m, n) \ge l \ge 0$).

a, *b*	REAL for stpqrt2 DOUBLE PRECISION for dtpqrt2 COMPLEX for ctpqrt2 COMPLEX*16 for ztpqrt2. Arrays: *a*, size (*lda*, *n*) contains the *n*-by-*n* upper triangular matrix *A*. *b*, size (*ldb*, *n*), the pentagonal *m*-by-*n* matrix *B*. The first (*m*-*l*) rows contain the rectangular *B1* matrix, and the next *l* rows contain the upper trapezoidal *B2* matrix.
lda	INTEGER. The leading dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The leading dimension of *b*; at least max(1, *m*).
ldt	INTEGER. The leading dimension of *t*; at least max(1, *n*).

Output Parameters

a	The elements on and above the diagonal of the array contain the upper triangular matrix *R*.
b	The pentagonal matrix *V*.
t	REAL for stpqrt2 DOUBLE PRECISION for dtpqrt2 COMPLEX for ctpqrt2 COMPLEX*16 for ztpqrt2. Array, size (*ldt*, *n*). The upper *n*-by-*n* upper triangular factor *T* of the block reflector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* < 0 and *info* = -*i*, the *i*th argument had an illegal value. If *info* = -1011, memory allocation error occurred.

?tprfb

Applies a real or complex "triangular-pentagonal" blocked reflector to a real or complex matrix, which is composed of two blocks.

Syntax

```
call stprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb, work, ldwork)
call dtprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb, work, ldwork)
call ctprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb, work, ldwork)
```

```
call ztprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb,
work, ldwork)
```

```
call tprfb(t, v, a, b[, direct][, storev][, side][, trans])
```

Include Files

- mkl.fi, lapack.f90

Description

The ?tprfb routine applies a real or complex "triangular-pentagonal" block reflector H, H^T, or H^H from either the left or the right to a real or complex matrix C, which is composed of two blocks A and B.

The block B is m-by-n. If *side* = 'R', A is m-by-k, and if *side* = 'L', A is of size k-by-n.

$$direct = \text{'F'} \quad direct = \text{'B'}$$

$$side = \text{'R'} \quad C = \begin{bmatrix} A & B \end{bmatrix} \quad C = \begin{bmatrix} B & A \end{bmatrix}$$

$$side = \text{'L'} \quad C = \begin{bmatrix} A \\ B \end{bmatrix} \quad C = \begin{bmatrix} B \\ A \end{bmatrix}$$

The pentagonal matrix V is composed of a rectangular block $V1$ and a trapezoidal block $V2$. The size of the trapezoidal block is determined by the parameter l, where $0 \le l \le k$. if $l=k$, the $V2$ block of V is triangular; if $l=0$, there is no trapezoidal block, thus $V = V1$ is rectangular.

	direct='F'	*direct*='B'
storev='C'	$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix}$ $V2$ is upper trapezoidal (first l rows of k-by-k upper triangular)	$V = \begin{bmatrix} V2 \\ V1 \end{bmatrix}$ $V2$ is lower trapezoidal (last l rows of k-by-k lower triangular matrix)
storev='R'	$V = \begin{bmatrix} V1 & V2 \end{bmatrix}$ $V2$ is lower trapezoidal (first l columns of k-by-k lower triangular matrix)	$V = \begin{bmatrix} V2 & V1 \end{bmatrix}$ $V2$ is upper trapezoidal (last l columns of k-by-k upper triangular matrix)

	side='L'	*side*='R'

storev='C'	*V* is *m*-by-*k* *V2* is *l*-by-*k*	*V* is *n*-by-*k* *V2* is *l*-by-*k*
storev='R'	*V* is *k*-by-*m* *V2* is *k*-by-*l*	*V* is *k*-by-*n* *V2* is *k*-by-*l*

Input Parameters

side CHARACTER*1.

= 'L': apply *H*, H^T, or H^H from the left,

= 'R': apply *H*, H^T, or H^H from the right.

trans CHARACTER*1.

= 'N': apply *H* (no transpose),

= 'T': apply H^T (transpose),

= 'C': apply H^H (conjugate transpose).

direct CHARACTER*1.

Indicates how *H* is formed from a product of elementary reflectors:

= 'F': *H* = *H*(1) *H*(2) . . . *H*(*k*) (Forward),

= 'B': *H* = *H*(*k*) . . . *H*(2) *H*(1) (Backward).

storev CHARACTER*1.

Indicates how the vectors that define the elementary reflectors are stored:

= 'C': Columns,

= 'R': Rows.

m INTEGER. The total number of rows in the matrix *B* (*m* ≥ 0).

n INTEGER. The number of columns in *B* (*n* ≥ 0).

k INTEGER. The order of the matrix *T*, which is the number of elementary reflectors whose product defines the block reflector. (*k* ≥ 0)

l INTEGER. The order of the trapezoidal part of *V*. (*k* ≥ *l* ≥ 0).

v REAL for stprfb

DOUBLE PRECISION for dtprfb

COMPLEX for ctprfb

COMPLEX*16 for ztprfb.

DIMENSION (*ldv*, *k*) if *storev* = 'C',

DIMENSION (*ldv*, *m*) if *storev* = 'R' and *side* = 'L',

DIMENSION (*ldv*, *n*) if *storev* = 'R' and *side* = 'R'.

The pentagonal matrix *V*, which contains the elementary reflectors *H*(1), *H*(2), ..., *H*(*k*).

ldv INTEGER. The leading dimension of the array *v*.

If *storev* = 'C' and *side* = 'L', at least max(1, *m*).

If *storev* = 'C' and *side* = 'R', at least max(1, *n*).

If *storev* = 'R' , at least *k*.

t REAL for stprfb

DOUBLE PRECISION for dtprfb

COMPLEX for ctprfb

COMPLEX*16 for ztprfb.

Array size (*ldt*, *k*). The triangular *k*-by-*k* matrix *T* in the representation of the block reflector.

ldt INTEGER. The leading dimension of the array *t* (*ldt* ≥ *k*).

a REAL for stprfb

DOUBLE PRECISION for dtprfb

COMPLEX for ctprfb

COMPLEX*16 for ztprfb.

DIMENSION (*lda*, *n*) if *side* = 'L',

DIMENSION (*lda*, *k*) if *side* = 'R'.

The *k*-by-*n* or *m*-by-*k* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.

If *side* = 'L', at least max(1, *k*).

If *side* = 'R', at least max(1, *m*).

b REAL for stprfb

DOUBLE PRECISION for dtprfb

COMPLEX for ctprfb

COMPLEX*16 for ztprfb.

Array size (*ldb*, *n*), the *m*-by-*n* matrix *B*.

ldb INTEGER. The leading dimension of the array *b* (*ldb* ≥ max(1, *m*)).

work REAL for stprfb

DOUBLE PRECISION for dtprfb

COMPLEX for ctprfb

COMPLEX*16 for ztprfb.

DIMENSION (*ldwork*, *n*) if *side* = 'L',

DIMENSION (*ldwork*, *k*) if *side* = 'R'.

Workspace array.

ldwork INTEGER. The leading dimension of the array *work*.

If *side* = 'L', at least *k*.

If *side* = 'R', at least *m*.

Output Parameters

a — Contains the corresponding block of $H*C$, H^T*C, H^H*C, $C*H$, $C*H^T$, or $C*H^H$.

b — Contains the corresponding block of $H*C$, H^T*C, H^H*C, $C*H$, $C*H^T$, or $C*H^H$.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?tpttf

Copies a triangular matrix from the standard packed format (TP) to the rectangular full packed format (TF).

Syntax

```
call stpttf( transr, uplo, n, ap, arf, info )
call dtpttf( transr, uplo, n, ap, arf, info )
call ctpttf( transr, uplo, n, ap, arf, info )
call ztpttf( transr, uplo, n, ap, arf, info )
```

Include Files

- mkl.fi

Description

The routine copies a triangular matrix *A* from the standard packed format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

Input Parameters

transr — CHARACTER*1.

= 'N': *arf* must be in the Normal format,

= 'T': *arf* must be in the Transpose format (for stpttf and dtpttf),

= 'C': *arf* must be in the Conjugate-transpose format (for ctpttf and ztpttf).

uplo — CHARACTER*1.

Specifies whether *A* is upper or lower triangular:

= 'U': A is upper triangular,

= 'L': A is lower triangular.

n — INTEGER. The order of the matrix *A*. $n \geq 0$.

ap — REAL for stpttf,

DOUBLE PRECISION for dtpttf,

COMPLEX for ctpttf,

DOUBLE COMPLEX for ztpttf.

Array, size at least max (1, *n**(*n*+1)/2).

On entry, the upper or lower triangular matrix *A*, packed columnwise in a linear array.

The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', *ap(i + (j-1)*j/2) = A(i,j)* for $1 \le i \le j$,

if *uplo* = 'L', *ap(i + (j-1)*(2n-j)/2) = A(i,j)* for $j \le i \le n$.

Output Parameters

arf

REAL for stpttf,

DOUBLE PRECISION for dtpttf,

COMPLEX for ctfttp,

DOUBLE COMPLEX for ztpttf.

Array, size at least *max* (1, *n**(*n*+1)/2).

On exit, the upper or lower triangular matrix *A* stored in the RFP format.

info

INTEGER.

=0: successful exit,

< 0: if *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?tpttr

Copies a triangular matrix from the standard packed format (TP) to the standard full format (TR) .

Syntax

```
call stpttr( uplo, n, ap, a, lda, info )
call dtpttr( uplo, n, ap, a, lda, info )
call ctpttr( uplo, n, ap, a, lda, info )
call ztpttr( uplo, n, ap, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine copies a triangular matrix *A* from the standard packed format to the standard full format.

Input Parameters

uplo — CHARACTER*1.

Specifies whether *A* is upper or lower triangular:

= 'U': A is upper triangular,

= 'L': A is lower triangular.

n — INTEGER. The order of the matrices *ap* and *a*. $n \geq 0$.

ap — REAL for stpttr,

DOUBLE PRECISION for dtpttr,

COMPLEX for ctpttr,

DOUBLE COMPLEX for ztpttr.

On entry, the upper or lower triangular matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', $ap(i + (j-1)*j/2) = A(i,j)$ for $1 \leq i \leq j$,

if *uplo* = 'L', $ap(i + (j-1)*(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

lda — INTEGER. The leading dimension of the array *a*. $lda \geq max(1,n)$.

Output Parameters

a — REAL for stpttr,

DOUBLE PRECISION for dtpttr,

COMPLEX for ctpttr,

DOUBLE COMPLEX for ztpttr.

Array, size (*lda, **).

On exit, the triangular matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

info — INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?trttf

Copies a triangular matrix from the standard full format (TR) to the rectangular full packed format (TF).

Syntax

```
call strttf( transr, uplo, n, a, lda, arf, info )
call dtrttf( transr, uplo, n, a, lda, arf, info )
```

```
call ctrttf( transr, uplo, n, a, lda, arf, info )
call ztrttf( transr, uplo, n, a, lda, arf, info )
```

Include Files

- mkl.fi

Description

The routine copies a triangular matrix *A* from the standard full format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

Input Parameters

transr

CHARACTER*1.

= 'N': *arf* must be in the Normal format,

= 'T': *arf* must be in the Transpose format (for strttf and dtrttf),

= 'C': *arf* must be in the Conjugate-transpose format (for ctrttf and ztrttf).

uplo

CHARACTER*1.

Specifies whether *A* is upper or lower triangular:

= 'U': A is upper triangular,

= 'L': A is lower triangular.

n

INTEGER. The order of the matrix *A*. $n \geq 0$.

a

REAL for strttf,

DOUBLE PRECISION for dtrttf,

COMPLEX for ctrttf,

DOUBLE COMPLEX for ztrttf.

Array, size (*lda*, *n*).

On entry, the triangular matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced.

lda

INTEGER. The leading dimension of the array *a*. *lda* ≥ *max*(1,*n*).

Output Parameters

arf

REAL for strttf,

DOUBLE PRECISION for dtrttf,

COMPLEX for ctrttf,

DOUBLE COMPLEX for ztrttf.

Array, size at least *max* (1, *n**(*n*+1)/2).

On exit, the upper or lower triangular matrix *A* stored in the RFP format.

info — INTEGER. If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?trttp

Copies a triangular matrix from the standard full format (TR) to the standard packed format (TP) .

Syntax

```
call strttp( uplo, n, a, lda, ap, info )
call dtrttp( uplo, n, a, lda, ap, info )
call ctrttp( uplo, n, a, lda, ap, info )
call ztrttp( uplo, n, a, lda, ap, info )
```

Include Files

- mkl.fi

Description

The routine copies a triangular matrix *A* from the standard full format to the standard packed format.

Input Parameters

uplo — CHARACTER*1.

Specifies whether *A* is upper or lower triangular:

= 'U': A is upper triangular,

= 'L': A is lower triangular.

n — INTEGER. The order of the matrix *A*, $n \geq 0$.

a — REAL for strttp,

DOUBLE PRECISION for dtrttp,

COMPLEX for ctrttp,

DOUBLE COMPLEX for ztrttp.

Array, size (*lda*, *n*).

On entry, the triangular matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. The leading dimension of the array *a*. *lda* ≥ *max*(1,*n*).

Output Parameters

ap	`REAL` for `strttp`, `DOUBLE PRECISION` for `dtrttp`, `COMPLEX` for `ctrttp`, `DOUBLE COMPLEX` for `ztrttp`. Array, size at least *max* (1, *n**(*n*+1)/2). On exit, the upper or lower triangular matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows: if *uplo* = 'U', *ap(i + (j-1)*j/2) = A(i,j)* for $1 \le i \le j$, if *uplo* = 'L', *ap(i + (j-1)*(2n-j)/2) = A(i,j)* for $j \le i \le n$.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* < 0, the *i*-th parameter had an illegal value. If *info* = -1011, memory allocation error occurred.

?pstf2

Computes the Cholesky factorization with complete pivoting of a real symmetric or complex Hermitian positive semi-definite matrix.

Syntax

```
call spstf2( uplo, n, a, lda, piv, rank, tol, work, info )
call dpstf2( uplo, n, a, lda, piv, rank, tol, work, info )
call cpstf2( uplo, n, a, lda, piv, rank, tol, work, info )
call zpstf2( uplo, n, a, lda, piv, rank, tol, work, info )
```

Include Files

- `mkl.fi`

Description

The real flavors `spstf2` and `dpstf2` compute the Cholesky factorization with complete pivoting of a real symmetric positive semi-definite matrix *A*. The complex flavors `cpstf2` and `zpstf2` compute the Cholesky factorization with complete pivoting of a complex Hermitian positive semi-definite matrix *A*. The factorization has the form:

$P^T * A * P = U^T * U$, if *uplo* = 'U' for real flavors,

$P^T * A * P = U^H * U$, if *uplo* = 'U' for complex flavors,

$P^T * A * P = L * L^T$, if *uplo* = 'L' for real flavors,

$P^T * A * P = L * L^H$, if *uplo* = 'L' for complex flavors,

where *U* is an upper triangular matrix and *L* is lower triangular, and *P* is stored as vector *piv*.

This algorithm does not check that *A* is positive semi-definite. This version of the algorithm calls level 2 BLAS.

Input Parameters

uplo — CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric or Hermitian matrix *A* is stored:

= 'U': Upper triangular,

= 'L': Lower triangular.

n — INTEGER. The order of the matrix *A*. *n* ≥ 0.

a — REAL for spstf2,

DOUBLE PRECISION for dpstf2,

COMPLEX for cpstf2,

DOUBLE COMPLEX for zpstf2.

Array, DIMENSION (*lda, **).

On entry, the symmetric matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

tol — REAL for spstf2 and cpstf2,

DOUBLE PRECISION for dpstf2 and zpstf2.

A user-defined tolerance.

If *tol* < 0, *n*ulp*max(A(k,k))* will be used (*ulp* is the Unit in the Last Place, or Unit of Least Precision). The algorithm terminates at the (*k* - 1)-st step if the pivot is not greater than *tol*.

lda — INTEGER. The leading dimension of the matrix *A*. *lda* ≥ *max*(1,*n*).

work — REAL for spstf2 and cpstf2,

DOUBLE PRECISION for dpstf2 and zpstf2.

Workspace array, DIMENSION at least *max* (1, *2*n).*

Output Parameters

piv — INTEGER. Array. DIMENSION at least *max (1,n)*.

piv is such that the non-zero entries are *P* (*piv* (*k*), *k*) = 1.

a — On exit, if *info* = 0, the factor U or L from the Cholesky factorization stored the same way as the matrix *A* is stored on entry.

rank — INTEGER.

The rank of *A*, determined by the number of steps the algorithm completed.

info — INTEGER.

< 0: if *info* = -*k*, the *k*-th parameter had an illegal value,

=0: the algorithm completed successfully,

> 0: the matrix *A* is rank-deficient with the computed rank, returned in *rank*, or indefinite.

dlat2s

Converts a double-precision triangular matrix to a single-precision triangular matrix.

Syntax

```
call dlat2s( uplo, n, a, lda, sa, ldsa, info )
```

Include Files

- mkl.fi

Description

This routine converts a double-precision triangular matrix *A* to a single-precision triangular matrix *SA*. dlat2s checks that all the elements of *A* are between *-RMAX* and *RMAX*, where *RMAX* is the overflow for the single-precision arithmetic. If this condition is not met, the conversion is aborted and a flag is raised. The routine does no parameter checking.

Input Parameters

uplo CHARACTER*1.

Specifies whether the matrix *A* is upper or lower triangular:

= 'U': *A* is upper triangular,

= 'L': *A* is lower triangular.

n INTEGER. The number of rows and columns of the matrix *A*. $n \geq 0$.

a DOUBLE PRECISION.

Array, DIMENSION (*lda, **).

On entry, the *n*-by-*n* triangular matrix *A*.

lda INTEGER. The leading dimension of the array *a*. $lda \geq max(1,n)$.

ldsa INTEGER. The leading dimension of the array *sa*. $ldsa \geq max(1,n)$.

Output Parameters

sa REAL.

Array, DIMENSION (*ldsa, **).

Only the part of *sa* determined by *uplo* is referenced. On exit,

- if *info* = 0, the *n*-by-*n* triangular matrix *SA*,
- if *info* > 0, the content of the part of *sa* determined by *uplo* is unspecified.

info INTEGER.

=0: successful exit,

> 0: an element of the matrix *A* is greater than the single-precision overflow threshold; in this case, the content of the part of *sa* determined by *uplo* is unspecified on exit.

zlat2c

Converts a double complex triangular matrix to a complex triangular matrix.

Syntax

```
call zlat2c( uplo, n, a, lda, sa, ldsa, info )
```

Include Files

- mkl.fi

Description

This routine is declared in mkl_lapack.fi.

The routine converts a DOUBLE COMPLEX triangular matrix *A* to a COMPLEX triangular matrix *SA*. zlat2c checks that the real and complex parts of all the elements of *A* are between *-RMAX* and *RMAX*, where *RMAX* is the overflow for the single-precision arithmetic. If this condition is not met, the conversion is aborted and a flag is raised. The routine does no parameter checking.

Input Parameters

uplo

CHARACTER*1.

Specifies whether the matrix *A* is upper or lower triangular:

= 'U': *A* is upper triangular,

= 'L': *A* is lower triangular.

n

INTEGER. The number of rows and columns in the matrix *A*. $n \geq 0$.

a

DOUBLE COMPLEX.

Array, DIMENSION (*lda, **).

On entry, the *n*-by-*n* triangular matrix *A*.

lda

INTEGER. The leading dimension of the array *a*. $lda \geq max(1,n)$.

ldsa

INTEGER. The leading dimension of the array *sa*. $ldsa \geq max(1,n)$.

Output Parameters

sa

COMPLEX.

Array, DIMENSION (*ldsa, **).

Only the part of *sa* determined by *uplo* is referenced. On exit,

- if *info* = 0, the *n*-by-*n* triangular matrix *sa*,
- if *info* > 0, the content of the part of *sa* determined by *uplo* is unspecified.

info

INTEGER.

=0: successful exit,

> 0: the real or complex part of an element of the matrix *A* is greater than the single-precision overflow threshold; in this case, the content of the part of *sa* determined by *uplo* is unspecified on exit.

?lacp2

Copies all or part of a real two-dimensional array to a complex array.

Syntax

```
call clacp2( uplo, m, n, a, lda, b, ldb )
call zlacp2( uplo, m, n, a, lda, b, ldb )
```

Include Files

- mkl.fi

Description

The routine copies all or part of a real matrix *A* to another matrix *B*.

Input Parameters

uplo — CHARACTER*1.

Specifies the part of the matrix *A* to be copied to *B*.

If *uplo* = 'U', the upper triangular part of *A*;

if *uplo* = 'L', the lower triangular part of *A*.

Otherwise, all of the matrix *A* is copied.

m — INTEGER. The number of rows in the matrix *A* (*m*≥ 0).

n — INTEGER. The number of columns in *A* (*n*≥ 0).

a — REAL for clacp2

DOUBLE PRECISION for zlacp2

Array, size at least (*lda*, *n*), contains the *m*-by-*n* matrix *A*.

If *uplo* = 'U', only the upper triangle or trapezoid is accessed; if *uplo* = 'L', only the lower triangle or trapezoid is accessed.

lda — INTEGER. The leading dimension of *a*; *lda*≥ max(1, *m*) .

ldb — INTEGER. The leading dimension of the output array *b*; *ldb*≥ max(1, *m*).

Output Parameters

b — COMPLEX for clacp2

DOUBLE COMPLEX for zlacp2.

Array, size (*ldb*, *n*).

On exit, `B = A` in the locations specified by *uplo*.

info — `INTEGER`. If `info = 0`, the execution is successful.

If `info < 0`, the *i*-th parameter had an illegal value.

If `info = -1011`, memory allocation error occurred.

?la_gbamv

Performs a matrix-vector operation to calculate error bounds.

Syntax

```
call sla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
call dla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
call cla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
call zla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
```

Include Files

- `mkl.fi`

Description

The `?la_gbamv` function performs one of the matrix-vector operations defined as

`y := alpha*abs(A)*abs(x) + beta*abs(y),`

or

$y := alpha*abs(A)^T*abs(x) + beta*abs(y)$,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*n* matrix, with *kl* sub-diagonals and *ku* super-diagonals.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by `(n + 1)` times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

trans — `INTEGER`. Specifies the operation to be performed:

If `trans = 'BLAS_NO_TRANS'`, then `y := alpha*abs(A)*abs(x) + beta*abs(y)`

If `trans = 'BLAS_TRANS'`, then $y := alpha*abs(A^T)*abs(x) + beta*abs(y)$

If `trans = 'BLAS_CONJ_TRANS'`, then $y := alpha*abs(A^T)*abs(x) + beta*abs(y)$

The parameter is unchanged on exit.

m INTEGER. Specifies the number of rows of the matrix *A*.

The value of *m* must be at least zero. Unchanged on exit.

n INTEGER. Specifies the number of columns of the matrix *A*.

The value of *n* must be at least zero. Unchanged on exit.

kl INTEGER. Specifies the number of sub-diagonals within the band of *A*.

kl≥ 0.

ku INTEGER. Specifies the number of super-diagonals within the band of *A*.

ku≥ 0.

alpha REAL for sla_gbamv and cla_gbamv

DOUBLE PRECISION for dla_gbamv and zla_gbamv

Specifies the scalar *alpha*. Unchanges on exit.

ab REAL for sla_gbamv

DOUBLE PRECISION for dla_gbamv

COMPLEX for cla_gbamv

DOUBLE COMPLEX for zla_gbamv

Array, DIMENSION(*ldab*, *).

Before entry, the leading *m*-by-*n* part of the array *ab* must contain the matrix of coefficients. The second dimension of *ab* must be at least max(1,*n*). Unchanged on exit.

ldab INTEGER. Specifies the leading dimension of *ab* as declared in the calling (sub)program. The value of *ldab* must be at least max(1, *m*). Unchanged on exit.

x REAL for sla_gbamv

DOUBLE PRECISION for dla_gbamv

COMPLEX for cla_gbamv

DOUBLE COMPLEX for zla_gbamv

Array, DIMENSION

(1 + (*n* - 1)*abs(*incx*)) when *trans* = 'N' or 'n'

and at least

(1 + (*m* - 1)*abs(*incx*)) otherwise.

Before entry, the incremented array *x* must contain the vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*. *incx* must not be zero.

beta REAL for sla_gbamv and cla_gbamv

DOUBLE PRECISION for dla_gbamv and zla_gbamv

Specifies the scalar *beta*. When *beta* is zero, you do not need to set *y* on input.

y	REAL for sla_gbamv and cla_gbamv DOUBLE PRECISION for dla_gbamv and zla_gbamv Array, DIMENSION at least (1 +(*m* - 1)*abs(*incy*)) when *trans* = 'N' or 'n' and at least (1 +(*n* - 1)*abs(*incy*)) otherwise. Before entry with *beta* non-zero, the incremented array *y* must contain the vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero. Unchanged on exit.

Output Parameters

y	Updated vector *y*.

?la_gbrcond

Estimates the Skeel condition number for a general banded matrix.

Syntax

```
call sla_gbrcond( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, cmode, c, info, work, iwork )
call dla_gbrcond( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, cmode, c, info, work, iwork )
```

Include Files

- mkl.fi

Description

The function estimates the Skeel condition number of

op(*A*) * op2(*C*)

where

the *cmode* parameter determines op2 as follows:

cmode **Value**	**op2(C)**
1	*C*
0	*I*
-1	inv(*C*)

The Skeel condition number

cond(*A*) = norminf(|inv(*A*)||*A*|)

is computed by computing scaling factors *R* such that

diag(*R*)*A*op2(*C*)

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

trans

CHARACTER*1. Must be 'N' or 'T' or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form $A*X = B$.

If *trans* = 'T', the system has the form $A^T*X = B$.

If *trans* = 'C', the system has the form $A^H*X = B$.

n

INTEGER. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

kl

INTEGER. The number of subdiagonals within the band of *A*; $kl \geq 0$.

ku

INTEGER. The number of superdiagonals within the band of *A*; $ku \geq 0$.

ab, *afb*, *c*, *work*

REAL for sla_gbrcond

DOUBLE PRECISION for dla_gbrcond

Arrays:

ab(*ldab*,*) contains the original band matrix *A* stored in rows from 1 to *kl* + *ku* + 1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

ab(*ku*+1+*i*-*j*,*j*) = A(*i*,*j*)

for

max(1,*j*-*ku*) ≤*i*≤ min(*n*,*j*+*kl*)

afb(*ldafb*,*) contains details of the LU factorization of the band matrix *A*, as returned by ?gbtrf. *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1.

c, DIMENSION*n*. The vector *C* in the formula op(*A*) * op2(*C*).

work is a workspace array of DIMENSION (5**n*).

The second dimension of *ab* and *afb* must be at least max(1, *n*).

ldab

INTEGER. The leading dimension of the array *ab*. *ldab*≥*kl*+*ku*+1.

ldafb

INTEGER. The leading dimension of *afb*. *ldafb*≥2**kl*+*ku*+1.

ipiv

INTEGER.

Array with DIMENSION*n*. The pivot indices from the factorization *A* = *P*L*U* as computed by ?gbtrf. Row *i* of the matrix was interchanged with row *ipiv*(*i*).

cmode

INTEGER. Determines op2(*C*) in the formula op(*A*) * op2(*C*) as follows:

If *cmode* = 1, op2(*C*) = *C*.

If *cmode* = 0, op2(*C*) = *I*.

If *cmode* = -1, op2(*C*) = inv(*C*).

iwork

INTEGER. Workspace array with DIMENSION*n*.

Output Parameters

info	INTEGER. If *info* = 0, the execution is successful. If *i* > 0, the *i*-th parameter is invalid.

See Also

?gbtrf

?la_gbrcond_c

*Computes the infinity norm condition number of op(A)*inv(diag(c)) for general banded matrices.*

Syntax

```
call cla_gbrcond_c( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, c, capply, info, work, rwork )
call zla_gbrcond_c( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, c, capply, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

```
op(A) * inv(diag(c))
```

where the *c* is a REAL vector for cla_gbrcond_c and a DOUBLE PRECISION vector for zla_gbrcond_c.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If *trans* = 'N', the system has the form $A*X = B$ (No transpose) If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose) If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose)
n	INTEGER. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of *A*; $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of *A*; $ku \geq 0$.
ab, *afb*, *work*	COMPLEX for cla_gbrcond_c DOUBLE COMPLEX for zla_gbrcond_c Arrays: *ab*(*ldab*,*) contains the original band matrix *A* stored in rows from 1 to *kl* + *ku* + 1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

`ab(ku+1+i-j,j) = A(i,j)`

for

`max(1,j-ku) ≤i≤ min(n,j+kl)`

afb(*ldafb*,*) contains details of the LU factorization of the band matrix *A*, as returned by `?gbtrf`. *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1.

work is a workspace array of `DIMENSION` (5**n*).

The second dimension of *ab* and *afb* must be at least `max(1, n)`.

ldab — `INTEGER`. The leading dimension of the array *ab*. *ldab*≥*kl*+*ku*+1.

ldafb — `INTEGER`. The leading dimension of *afb*. *ldafb*≥2**kl*+*ku*+1.

ipiv — `INTEGER`.

Array with `DIMENSION`*n*. The pivot indices from the factorization *A = P*L*U* as computed by `?gbtrf`. Row *i* of the matrix was interchanged with row *ipiv*(*i*).

c, *rwork* — `REAL` for `cla_gbrcond_c`

`DOUBLE PRECISION` for `zla_gbrcond_c`

Array *c* with `DIMENSION`*n*. The vector *c* in the formula

`op(A) * inv(diag(c))`.

Array *rwork* with `DIMENSION`*n* is a workspace.

capply — `LOGICAL`. If `.TRUE.`, then the function uses the vector *c* from the formula

`op(A) * inv(diag(c))`.

Output Parameters

info — `INTEGER`.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

`?gbtrf`

?la_gbrcond_x

*Computes the infinity norm condition number of op(A)*diag(x) for general banded matrices.*

Syntax

```
call cla_gbrcond_x( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, x, info, work, rwork )
call zla_gbrcond_x( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, x, info, work, rwork )
```

Include Files

- `mkl.fi`

Description

The function computes the infinity norm condition number of

`op(A) * diag(x)`

where the *x* is a `COMPLEX` vector for `cla_gbrcond_x` and a `DOUBLE COMPLEX` vector for `zla_gbrcond_x`.

Input Parameters

trans

`CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Specifies the form of the system of equations:

If *trans* = `'N'`, the system has the form $A*X = B$ (No transpose)

If *trans* = `'T'`, the system has the form $A^T*X = B$ (Transpose)

If *trans* = `'C'`, the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose)

n

`INTEGER`. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

kl

`INTEGER`. The number of subdiagonals within the band of *A*; $kl \geq 0$.

ku

`INTEGER`. The number of superdiagonals within the band of *A*; $ku \geq 0$.

ab, *afb*, *x*, *work*

`COMPLEX` for `cla_gbrcond_x`

`DOUBLE COMPLEX` for `zla_gbrcond_x`

Arrays:

ab(*ldab*,*) contains the original band matrix *A* stored in rows from 1 to *kl* + *ku* + 1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

ab(*ku*+1+*i*-*j*,*j*) = A(*i*,*j*)

for

max(1,*j*-*ku*) ≤*i*≤ min(*n*,*j*+*kl*)

afb(*ldafb*,*) contains details of the LU factorization of the band matrix *A*, as returned by `?gbtrf`. *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1.

x, `DIMENSION`*n*. The vector *x* in the formula `op(A) * diag(x)`.

work is a workspace array of `DIMENSION` (2**n*).

The second dimension of *ab* and *afb* must be at least `max(1, n)`.

ldab

`INTEGER`. The leading dimension of the array *ab*. *ldab*≥*kl*+*ku*+1.

ldafb

`INTEGER`. The leading dimension of *afb*. *ldafb*≥2**kl*+*ku*+1.

ipiv

`INTEGER`.

Array with `DIMENSION`*n*. The pivot indices from the factorization *A = P*L*U* as computed by `?gbtrf`. Row *i* of the matrix was interchanged with row *ipiv*(*i*).

rwork

`REAL` for `cla_gbrcond_x`

`DOUBLE PRECISION` for `zla_gbrcond_x`

Array *rwork* with DIMENSION*n* is a workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?gbtrf

?la_gbrfsx_extended

Improves the computed solution to a system of linear equations for general banded matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

```
call sla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb, ldafb,
ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res,
ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call dla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb, ldafb,
ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res,
ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call cla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb, ldafb,
ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res,
ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call zla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb, ldafb,
ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res,
ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

Include Files

- mkl.fi

Description

The ?la_gbrfsx_extended subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The ?gbrfsx routine calls ?la_gbrfsx_extended to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

Use ?la_gbrfsx_extended to set only the second fields of *err_bnds_norm* and *err_bnds_comp*.

Input Parameters

prec_type INTEGER.

Specifies the intermediate precision to be used in refinement. The value is defined by ilaprec(p), where p is a CHARACTER and:

If *p* = 'S': Single.

If *p* = 'D': Double.

If *p* = 'I': Indigenous.

If *p* = 'X', 'E': Extra.

trans_type INTEGER.

Specifies the transposition operation on *A*. The value is defined by ilatrans(t), where t is a CHARACTER and:

If *t* = 'N': No transpose.

If *t* = 'T': Transpose.

If *t* = 'C': Conjugate Transpose.

n INTEGER. The number of linear equations; the order of the matrix *A*; *n*≥ 0.

kl INTEGER. The number of subdiagonals within the band of *A*; *kl*≥ 0.

ku INTEGER. The number of superdiagonals within the band of *A*; *ku*≥ 0.

nrhs INTEGER. The number of right-hand sides; the number of columns of the matrix *B*.

ab, *afb*, *b*, *y* REAL for sla_gbrfsx_extended

DOUBLE PRECISION for dla_gbrfsx_extended

COMPLEX for cla_gbrfsx_extended

DOUBLE COMPLEX for zla_gbrfsx_extended.

Arrays: *ab*(*ldab*,*), *afb*(*ldafb*,*), *b*(*ldb*,*), *y*(*ldy*,*).

The array *ab* contains the original *n*-by-*n* matrix *A*. The second dimension of *ab* must be at least max(1,*n*).

The array *afb* contains the factors *L* and *U* from the factorization *A = P*L*U*) as computed by ?gbtrf. The second dimension of *afb* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

The array *y* on entry contains the solution matrix *X* as computed by ?gbtrs. The second dimension of *y* must be at least max(1,*nrhs*).

ldab INTEGER. The leading dimension of the array *ab*; *ldab*≥ max(1,*n*).

ldafb INTEGER. The leading dimension of the array *afb*; *ldafb*≥ max(1,*n*).

ipiv INTEGER.

Array, DIMENSION at least max(1, *n*). Contains the pivot indices from the factorization *A = P*L*U*) as computed by ?gbtrf; row *i* of the matrix was interchanged with row *ipiv*(*i*).

colequ LOGICAL. If *colequ* = .TRUE., column equilibration was done to *A* before calling this routine. This is needed to compute the solution and error bounds correctly.

c

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

c contains the column scale factors for *A*. If *colequ* = .FALSE., *c* is not accessed.

If *c* is input, each element of *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

ldy

INTEGER. The leading dimension of the array *y*; *ldy*≥ max(1, *n*).

n_norms

INTEGER. Determines which error bounds to return. See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

If *n_norms*≥ 1, returns normwise error bounds.

If *n_norms*≥ 2, returns componentwise error bounds.

err_bnds_norm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION(*nrhs*,*n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left|Xtrue_{ji} - X_{ji}\right|}{\max_j \left|X_{ji}\right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and

sqrt(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3 Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are `1/(norm(1/z,inf)*norm(z,inf))` for some appropriately scaled matrix *Z*.

Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

Use this subroutine to set only the second field above.

err_bnds_comp `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, `DIMENSION(`*nrhs*`,`*n_err_bnds*`)`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is nit requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the follwoing three fields:

err=1 "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.

	err=2	"Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
	err=3	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are 1/(norm(1/z,inf)*norm(z,inf)) for some appropriately scaled matrix *Z*. Let *z*=*s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1. Use this subroutine to set only the second field above.
res, *dy*, *y_tail*	REAL for sla_gbrfsx_extended DOUBLE PRECISION for dla_gbrfsx_extended COMPLEX for cla_gbrfsx_extended DOUBLE COMPLEX for zla_gbrfsx_extended. Workspace arrays of DIMENSION*n*. *res* holds the intermediate residual. *dy* holds the intermediate solution. *y_tail* holds the trailing bits of the intermediate solution.	
ayb	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Workspace array, DIMENSION*n*.	
rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.	

ithresh INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

rthresh REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

norm(dx_{*i*+1}) < *rthresh* * norm(dx_*i*)

where norm(*z*) is the infinity norm of *Z*.

rthresh satisfies

0 < *rthresh*≤ 1.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution *y* is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise LOGICAL

If .TRUE., the function ignores componentwise convergence. Default value is .FALSE.

Output Parameters

y REAL for sla_gbrfsx_extended

DOUBLE PRECISION for dla_gbrfsx_extended

COMPLEX for cla_gbrfsx_extended

DOUBLE COMPLEX for zla_gbrfsx_extended.

The improved solution matrix *Y*.

berr_out REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least max(1, *nrhs*). Contains the componentwise relative backward error for right-hand-side *j* from the formula

max(*i*) (abs(*res*(*i*)) / (abs(op(*A*))*abs(*y*) + abs(*B*))(*i*))

where abs(*z*) is the componentwise absolute value of the matrix or vector *Z*. This is computed by ?la_lin_berr.

err_bnds_norm, *err_bnds_comp*	Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array `( 1:nrhs, 2 )`. The other elements are kept unchanged.
info	`INTEGER`. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed. If *info* = -*i*, the *i*-th parameter had an illegal value.

See Also

?gbrfsx
?gbtrf
?gbtrs
?lamch
ilaprec
ilatrans
?la_lin_berr

?la_gbrpvgrw

Computes the reciprocal pivot growth factor `norm(A)/ norm(U)` *for a general band matrix.*

Syntax

```
call sla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
call dla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
call cla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
call zla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
```

Include Files

- mkl.fi

Description

The `?la_gbrpvgrw` routine computes the reciprocal pivot growth factor `norm(A)/norm(U)`. The *max absolute element* norm is used. If this is much less than 1, the stability of the *LU* factorization of the equilibrated matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

n	`INTEGER`. The number of linear equations, the order of the matrix *A*; $n \geq 0$.
kl	`INTEGER`. The number of subdiagonals within the band of *A*; $kl \geq 0$.
ku	`INTEGER`. The number of superdiagonals within the band of *A*; $ku \geq 0$.
ncols	`INTEGER`. The number of columns of the matrix *A*; $ncols \geq 0$.

ab, *afb*	`REAL` for `sla_gbrpvgrw`
	`DOUBLE PRECISION` for `dla_gbrpvgrw`
	`COMPLEX` for `cla_gbrpvgrw`

DOUBLE COMPLEX for zla_gbrpvgrw.

Arrays: *ab*(*ldab*,*), *afb*(*ldafb*,*).

ab contains the original band matrix *A* (see Matrix Storage Schemes) stored in rows from 1 to *kl* + *ku* + 1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

ab(*ku*+1+*i*-*j*,*j*) = A(*i*,*j*)

for

max(1,*j*-*ku*) ≤*i*≤ min(*n*,*j*+*kl*)

afb contains details of the LU factorization of the band matrix *A*, as returned by ?gbtrf. *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1.

ldab — INTEGER. The leading dimension of *ab*; *ldab*≥*kl*+*ku*+1.

ldafb — INTEGER. The leading dimension of *afb*; *ldafb*≥ 2**kl*+*ku*+1.

See Also

?gbtrf

?la_geamv

Computes a matrix-vector product using a general matrix to calculate error bounds.

Syntax

```
call sla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call dla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call cla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call zla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

Include Files

- mkl.fi

Description

The ?la_geamv routines perform a matrix-vector operation defined as

y := *alpha**abs(*A*)*(*x*) + *beta**abs(*y*),

or

y := *alpha**abs(A^T)*abs(*x*) + *beta**abs(*y*),

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*n* matrix.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by `(n + 1)` times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

trans

`CHARACTER*1`. Specifies the operation:

if *trans* = `BLAS_NO_TRANS` , then $y := alpha*abs(A)*abs(x) + beta*abs(y)$

if *trans* = `BLAS_TRANS`, then $y := alpha*abs(A^T)*abs(x) + beta*abs(y)$

if *trans* = `'BLAS_CONJ_TRANS`, then $y := alpha*abs(A^T)*abs(x) + beta*abs(y)$.

m

`INTEGER`. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.

n

`INTEGER`. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.

alpha

`REAL` for `sla_geamv` and for `cla_geamv`

`DOUBLE PRECISION` for `dla_geamv` and `zla_geamv`

Specifies the scalar *alpha*.

a

`REAL` for `sla_geamv`

`DOUBLE PRECISION` for `dla_geamv`

`COMPLEX` for `cla_geamv`

`DOUBLE COMPLEX` for `zla_geamv`

Array, `DIMENSION(lda, *)`. Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients. The second dimension of *a* must be at least `max(1,n)`.

lda

`INTEGER`. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, m)`.

x

`REAL` for `sla_geamv`

`DOUBLE PRECISION` for `dla_geamv`

`COMPLEX` for `cla_geamv`

`DOUBLE COMPLEX` for `zla_geamv`

Array, `DIMENSION` at least `(1+(n-1)*abs(incx))` when *trans* = `'N'` or `'n'` and at least `(1+(m - 1)*abs(incx))` otherwise. Before entry, the incremented array *x* must contain the vector *X*.

incx

`INTEGER`. Specifies the increment for the elements of *x*.

The value of *incx* must be non-zero.

beta

`REAL` for `sla_geamv` and for `cla_geamv`

DOUBLE PRECISION for dla_geamv and zla_geamv

Specifies the scalar *beta*. When *beta* is zero, you do not need to set *y* on input.

y

REAL for sla_geamv and for cla_geamv

DOUBLE PRECISION for dla_geamv and zla_geamv

Array, DIMENSION at least (1 +(*m* - 1)*abs(*incy*)) when *trans* = 'N' or 'n' and at least (1 +(*n* - 1)*abs(*incy*)) otherwise. Before entry with non-zero *beta*, the incremented array *y* must contain the vector *Y*.

incy

INTEGER. Specifies the increment for the elements of *y*.

The value of *incy* must be non-zero.

Output Parameters

y

Updated vector *Y*.

?la_gercond

Estimates the Skeel condition number for a general matrix.

Syntax

```
call sla_gercond( trans, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
call dla_gercond( trans, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
```

Include Files

- mkl.fi

Description

The function estimates the Skeel condition number of

op(*A*) * op2(*C*)

where

the *cmode* parameter determines op2 as follows:

cmode **Value**	**op2(C)**
1	*C*
0	*I*
-1	inv(*C*)

The Skeel condition number

cond(*A*) = norminf(|inv(*A*)||*A*|

is computed by computing scaling factors *R* such that

diag(*R*)*A*op2(*C*)

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

trans — CHARACTER*1. Must be 'N' or 'T' or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form $A*X = B$ (No transpose).

If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose).

If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose).

n — INTEGER. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

a, *af*, *c*, *work* — REAL for sla_gercond

DOUBLE PRECISION for dla_gercond

Arrays:

a(*lda*,*) contains the original general *n*-by-*n* matrix *A*.

af(*ldaf*,*) contains factors *L* and *U* from the factorization of the general matrix *A*=*P*L*U*, as returned by ?getrf.

c, DIMENSION*n*. The vector *C* in the formula op(*A*) * op2(*C*).

work is a workspace array of DIMENSION (3**n*).

The second dimension of *a* and *af* must be at least max(1, *n*).

lda — INTEGER. The leading dimension of the array *a*. *lda*≥max(1,*n*).

ldaf — INTEGER. The leading dimension of *af*. *ldaf*≥max(1, *n*).

ipiv — INTEGER.

Array with DIMENSION*n*. The pivot indices from the factorization *A* = *P*L*U* as computed by ?getrf. Row *i* of the matrix was interchanged with row *ipiv*(*i*).

cmode — INTEGER. Determines op2(*C*) in the formula op(*A*) * op2(*C*) as follows:

If *cmode* = 1, op2(*C*) = *C*.

If *cmode* = 0, op2(*C*) = *I*.

If *cmode* = -1, op2(*C*) = inv(*C*).

iwork — INTEGER. Workspace array with DIMENSION*n*.

Output Parameters

info — INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?getrf

?la_gercond_c

*Computes the infinity norm condition number of op(A)*inv(diag(c)) for general matrices.*

Syntax

```
call cla_gercond_c( trans, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
call zla_gercond_c( trans, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

op(*A*) * inv(diag(c))

where the *c* is a REAL vector for cla_gercond_c and a DOUBLE PRECISION vector for zla_gercond_c.

Input Parameters

trans

CHARACTER*1. Must be 'N' or 'T' or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form $A*X = B$ (No transpose)

If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose)

If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose)

n

INTEGER. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

a, *af*, *work*

COMPLEX for cla_gercond_c

DOUBLE COMPLEX for zla_gercond_c

Arrays:

a(*lda*,*) contains the original general *n*-by-*n* matrix *A*.

af(*ldaf*,*) contains the factors *L* and *U* from the factorization *A=P*L*U* as returned by ?getrf.

work is a workspace array of DIMENSION (2**n*).

The second dimension of *a* and *af* must be at least max(1, *n*).

lda

INTEGER. The leading dimension of the array *a*. $lda \geq \max(1,n)$.

ldaf

INTEGER. The leading dimension of *af*. $ldaf \geq \max(1,n)$.

ipiv

INTEGER.

Array with DIMENSION*n*. The pivot indices from the factorization *A = P*L*U* as computed by ?getrf. Row *i* of the matrix was interchanged with row *ipiv*(*i*).

c, *rwork*

REAL for cla_gercond_c

DOUBLE PRECISION for zla_gercond_c

Array *c* with DIMENSION*n*. The vector *c* in the formula

op(*A*) * inv(diag(*c*)).

Array *rwork* with DIMENSION*n* is a workspace.

capply LOGICAL. If *capply*=.TRUE., then the function uses the vector *c* from the formula

op(*A*) * inv(diag(*c*)).

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?getrf

?la_gercond_x

*Computes the infinity norm condition number of op(A)*diag(x) for general matrices.*

Syntax

call cla_gercond_x(*trans*, *n*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *x*, *info*, *work*, *rwork*)

call zla_gercond_x(*trans*, *n*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *x*, *info*, *work*, *rwork*)

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

op(*A*) * diag(*x*)

where the *x* is a COMPLEX vector for cla_gercond_x and a DOUBLE COMPLEX vector for zla_gercond_x.

Input Parameters

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form $A*X = B$ (No transpose)

If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose)

If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose)

n INTEGER. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

a, *af*, *x*, *work* COMPLEX for cla_gercond_x

DOUBLE COMPLEX for zla_gercond_x

Arrays:

a(*lda*,*) contains the original general *n*-by-*n* matrix *A*.

af(*ldaf*,*) contains the factors *L* and *U* from the factorization *A=P*L*U* as returned by ?getrf.

x, DIMENSION*n*. The vector *x* in the formula op(*A*) * diag(*x*).

work is a workspace array of DIMENSION (2**n*).

The second dimension of *a* and *af* must be at least max(1, *n*).

lda INTEGER. The leading dimension of the array *a*. *lda*≥max(1,*n*).

ldaf INTEGER. The leading dimension of *af*. *ldaf*≥max(1,*n*).

ipiv INTEGER.

Array with DIMENSION*n*. The pivot indices from the factorization *A = P*L*U* as computed by ?getrf. Row *i* of the matrix was interchanged with row *ipiv*(*i*).

rwork REAL for cla_gercond_x

DOUBLE PRECISION for zla_gercond_x

Array *rwork* with DIMENSION*n* is a workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?getrf

?la_gerfsx_extended

Improves the computed solution to a system of linear equations for general matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

call sla_gerfsx_extended(*prec_type*, *trans_type*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *errs_n*, *errs_c*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call dla_gerfsx_extended(*prec_type*, *trans_type*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *errs_n*, *errs_c*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call cla_gerfsx_extended(*prec_type*, *trans_type*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *errs_n*, *errs_c*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call zla_gerfsx_extended(*prec_type*, *trans_type*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *errs_n*, *errs_c*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

Include Files

- mkl.fi

Description

The ?la_gerfsx_extended subroutine improves the computed solution to a system of linear equations for general matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The ?gerfsx routine calls ?la_gerfsx_extended to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for *errs_n* and *errs_c* for details of the error bounds.

Use ?la_gerfsx_extended to set only the second fields of *errs_n* and *errs_c*.

Input Parameters

prec_type

INTEGER.

Specifies the intermediate precision to be used in refinement. The value is defined by ilaprec(p), where p is a CHARACTER and:

If *p* = 'S': Single.

If *p* = 'D': Double.

If *p* = 'I': Indigenous.

If *p* = 'X', 'E': Extra.

trans_type

INTEGER.

Specifies the transposition operation on *A*. The value is defined by ilatrans(t), where t is a CHARACTER and:

If *t* = 'N': No transpose.

If *t* = 'T': Transpose.

If *t* = 'C': Conjugate Transpose.

n

INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrix *B*.

a, *af*, *b*, *y*

REAL for sla_gerfsx_extended

DOUBLE PRECISION for dla_gerfsx_extended

COMPLEX for cla_gerfsx_extended

DOUBLE COMPLEX for zla_gerfsx_extended.

Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *y*(*ldy*,*).

The array *a* contains the original matrix *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1,*n*).

The array *af* contains the factors *L* and *U* from the factorization *A = P*L*U*) as computed by ?getrf. The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

The array *y* on entry contains the solution matrix *X* as computed by ?getrs. The second dimension of *y* must be at least max(1,*nrhs*).

lda INTEGER. The leading dimension of the array *a*; *lda*≥ max(1,*n*).

ldaf INTEGER. The leading dimension of the array *af*; *ldaf*≥ max(1,*n*).

ipiv INTEGER.

Array, DIMENSION at least max(1, *n*). Contains the pivot indices from the factorization *A = P*L*U*) as computed by ?getrf; row *i* of the matrix was interchanged with row *ipiv*(*i*).

colequ LOGICAL. If *colequ* = .TRUE., column equilibration was done to *A* before calling this routine. This is needed to compute the solution and error bounds correctly.

c REAL for single precision flavors (sla_gerfsx_extended, cla_gerfsx_extended)

DOUBLE PRECISION for double precision flavors (dla_gerfsx_extended, zla_gerfsx_extended).

c contains the column scale factors for *A*. If *colequ* = .FALSE., *c* is not used.

If *c* is input, each element of *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

ldy INTEGER. The leading dimension of the array *y*; *ldy*≥ max(1, *n*).

n_norms INTEGER. Determines which error bounds to return. See *errs_n* and *errs_c* descriptions in *Output Arguments* section below.

If *n_norms*≥ 1, returns normwise error bounds.

If *n_norms*≥ 2, returns componentwise error bounds.

errs_n REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION(*nrhs*,*n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j \left| Xtrue_{ji} - X_{ji} \right|}{\max_j \left| X_{ji} \right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *errs_n(i,:)* corresponds to the *i*-th right-hand side.

The second index in *errs_n(:,err)* contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.
err=2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are 1/(norm(1/z,inf)*norm(z,inf)) for some appropriately scaled matrix *Z*. Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1. Use this subroutine to set only the second field above.

errs_c

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION(*nrhs*,*n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is nit requested (*params(3)* = 0.0), then *errs_c* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *errs_c(i,:)* corresponds to the *i*-th right-hand side.

The second index in *errs_c(:,err)* contains the follwoing three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors.
err=2	"Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for single precision flavors and *sqrt*(*n*)**dlamch*(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are `1/(norm(1/z,inf)*norm(z,inf))` for some appropriately scaled matrix *Z*. Let *z=s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1. Use this subroutine to set only the second field above.

res, *dy*, *y_tail*

`REAL` for `sla_gerfsx_extended`

`DOUBLE PRECISION` for `dla_gerfsx_extended`

`COMPLEX` for `cla_gerfsx_extended`

`DOUBLE COMPLEX` for `zla_gerfsx_extended`.

Workspace arrays of DIMENSION*n*.

res holds the intermediate residual.

dy holds the intermediate solution.

y_tail holds the trailing bits of the intermediate solution.

ayb

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION*n*.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in *errs_n* and *errs_c* may no longer be trustworthy.

rthresh

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

norm(dx_{*i*+1}) < *rthresh* * norm(dx_*i*)

where norm(*z*) is the infinity norm of *Z*.

rthresh satisfies

0 < *rthresh*≤ 1.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution *y* is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

LOGICAL

If .TRUE., the function ignores componentwise convergence. Default value is .FALSE.

Output Parameters

y	REAL for sla_gerfsx_extended DOUBLE PRECISION for dla_gerfsx_extended COMPLEX for cla_gerfsx_extended DOUBLE COMPLEX for zla_gerfsx_extended. The improved solution matrix *Y*.
berr_out	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least max(1, *nrhs*). Contains the componentwise relative backward error for right-hand-side *j* from the formula max(*i*) (abs(*res*(*i*)) / (abs(op(*A*))*abs(*y*) + abs(*B*))(*i*)) where abs(*z*) is the componentwise absolute value of the matrix or vector *Z*. This is computed by ?la_lin_berr.
errs_n, *errs_c*	Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array (1:*nrhs*, 2). The other elements are kept unchanged.
info	INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed. If *info* = -*i*, the *i*-th parameter had an illegal value.

See Also

?gerfsx
?getrf
?getrs
?lamch
ilaprec
ilatrans
?la_lin_berr

?la_heamv

Computes a matrix-vector product using a Hermitian indefinite matrix to calculate error bounds.

Syntax

```
call cla_heamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call zla_heamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

Include Files

- mkl.fi

Description

The ?la_heamv routines perform a matrix-vector operation defined as

y := *alpha**abs(*A*)*abs(*x*) + *beta**abs(*y*),

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *n*-by-*n* Hermitian matrix.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by `(n + 1)` times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

uplo

`CHARACTER*1`.

Specifies whether the upper or lower triangular part of the array A is to be referenced:

If *uplo* = `'BLAS_UPPER'`, only the upper triangular part of A is to be referenced,

If *uplo* = `'BLAS_LOWER'`, only the lower triangular part of A is to be referenced.

n

`INTEGER`. Specifies the number of rows and columns of the matrix *A*. The value of *n* must be at least zero.

alpha

`REAL` for `cla_heamv`

`DOUBLE PRECISION` for `zla_heamv`

Specifies the scalar *alpha*.

a

`COMPLEX` for `cla_heamv`

`DOUBLE COMPLEX` for `zla_heamv`

Array, `DIMENSION(lda, *)`. Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients. The second dimension of *a* must be at least `max(1,n)`.

lda

`INTEGER`. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, n)`.

x

`COMPLEX` for `cla_heamv`

`DOUBLE COMPLEX` for `zla_heamv`

Array, `DIMENSION` at least `(1+(n-1)*abs(incx))`. Before entry, the incremented array *x* must contain the vector *X*.

incx

`INTEGER`. Specifies the increment for the elements of *x*.

The value of *incx* must be non-zero.

beta

`REAL` for `cla_heamv`

`DOUBLE PRECISION` for `zla_heamv`

Specifies the scalar *beta*. When *beta* is zero, you do not need to set *y* on input.

y	REAL for cla_heamv DOUBLE PRECISION for zla_heamv Array, DIMENSION at least (1 +(*n* – 1)*abs(*incy*)) otherwise. Before entry with non-zero *beta*, the incremented array *y* must contain the vector *Y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must be non-zero.

Output Parameters

y	Updated vector *Y*.

?la_hercond_c

*Computes the infinity norm condition number of op(A)*inv(diag(c)) for Hermitian indefinite matrices.*

Syntax

```
call cla_hercond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
call zla_hercond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

op(*A*) * inv(diag(c))

where the *c* is a REAL vector for cla_hercond_c and a DOUBLE PRECISION vector for zla_hercond_c.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If *uplo* = 'U', the upper triangle of *A* is stored, If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The number of linear equations, that is, the order of the matrix *A*; *n*≥ 0.
a	COMPLEX for cla_hercond_c DOUBLE COMPLEX for zla_hercond_c Array, DIMENSION(*lda*, *). On entry, the *n*-by-*n* matrix A. The second dimension of *a* must be at least max(1,*n*).
lda	INTEGER. The leading dimension of the array *a*. *lda*≥max(1,*n*).
af	COMPLEX for cla_hercond_c

DOUBLE COMPLEX for zla_hercond_c

Array, DIMENSION(*ldaf*, *). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?hetrf. The second dimension of *af* must be at least max(1,*n*).

ldaf — INTEGER. The leading dimension of the array *af*. *ldaf*≥max(1,*n*).

ipiv — INTEGER.

Array with DIMENSION*n*. Details of the interchanges and the block structure of D as determined by ?hetrf.

c — REAL for cla_hercond_c

DOUBLE PRECISION for zla_hercond_c

Array *c* with DIMENSION*n*. The vector *c* in the formula

op(*A*) * inv(diag(*c*)).

capply — LOGICAL. If .TRUE., then the function uses the vector *c* from the formula

op(*A*) * inv(diag(*c*)).

work — COMPLEX for cla_hercond_c

DOUBLE COMPLEX for zla_hercond_c

Array DIMENSION 2**n*. Workspace.

rwork — REAL for cla_hercond_c

DOUBLE PRECISION for zla_hercond_c

Array DIMENSION*n*. Workspace.

Output Parameters

info — INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?hetrf

?la_hercond_x

*Computes the infinity norm condition number of op(A)*diag(x) for Hermitian indefinite matrices.*

Syntax

```
call cla_hercond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
call zla_hercond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

`op(A) * diag(x)`

where the *x* is a `COMPLEX` vector for `cla_hercond_x` and a `DOUBLE COMPLEX` vector for `zla_hercond_x`.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Specifies the triangle of A to store:

If *uplo* = `'U'`, the upper triangle of *A* is stored,

If *uplo* = `'L'`, the lower triangle of *A* is stored.

n

`INTEGER`. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

a

`COMPLEX` for `cla_hercond_c`

`DOUBLE COMPLEX` for `zla_hercond_c`

Array, `DIMENSION(lda, *)`. On entry, the *n*-by-*n* matrix A. The second dimension of *a* must be at least `max(1,n)`.

lda

`INTEGER`. The leading dimension of the array *a*. $lda \geq \max(1,n)$.

af

`COMPLEX` for `cla_hercond_c`

`DOUBLE COMPLEX` for `zla_hercond_c`

Array, `DIMENSION(ldaf, *)`. The block diagonal matrix D and the multipliers used to obtain the factor `U` or `L` as computed by `?hetrf`. The second dimension of *af* must be at least `max(1,n)`.

ldaf

`INTEGER`. The leading dimension of the array *af*. $ldaf \geq \max(1,n)$.

ipiv

`INTEGER`.

Array with `DIMENSION`*n*. Details of the interchanges and the block structure of D as determined by `?hetrf`.

x

`COMPLEX` for `cla_hercond_c`

`DOUBLE COMPLEX` for `zla_hercond_c`

Array *x* with `DIMENSION`*n*. The vector *x* in the formula

`op(A) * inv(diag(x))`.

work

`COMPLEX` for `cla_hercond_c`

`DOUBLE COMPLEX` for `zla_hercond_c`

Array `DIMENSION` 2**n*. Workspace.

rwork

`REAL` for `cla_hercond_c`

`DOUBLE PRECISION` for `zla_hercond_c`

Array `DIMENSION`*n*. Workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?hetrf

?la_herfsx_extended

Improves the computed solution to a system of linear equations for Hermitian indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

call cla_herfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call zla_herfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

Include Files

- mkl.fi

Description

The ?la_herfsx_extended subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The ?herfsx routine calls ?la_herfsx_extended to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

Use ?la_herfsx_extended to set only the second fields of *err_bnds_norm* and *err_bnds_comp*.

Input Parameters

prec_type INTEGER.

Specifies the intermediate precision to be used in refinement. The value is defined by ilaprec(p), where *p* is a CHARACTER and:

If *p* = 'S': Single.

If *p* = 'D': Double.

If *p* = 'I': Indigenous.

If *p* = 'X', 'E': Extra.

uplo CHARACTER*1. Must be 'U' or 'L'.

Specifies the triangle of A to store:

If *uplo* = 'U', the upper triangle of *A* is stored,

If *uplo* = 'L', the lower triangle of *A* is stored.

n — INTEGER. The number of linear equations; the order of the matrix *A*; *n*≥ 0.

nrhs — INTEGER. The number of right-hand sides; the number of columns of the matrix *B*.

a, af, b, y — COMPLEX for cla_herfsx_extended

DOUBLE COMPLEX for zla_herfsx_extended.

Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *y*(*ldy*,*).

The array *a* contains the original *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1,*n*).

The array *af* contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?hetrf. The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the right-hand-side of the matrix *B*. The second dimension of *b* must be at least max(1,*nrhs*).

The array *y* on entry contains the solution matrix *X* as computed by ?hetrs. The second dimension of *y* must be at least max(1,*nrhs*).

lda — INTEGER. The leading dimension of the array *a*; *lda*≥ max(1,*n*).

ldaf — INTEGER. The leading dimension of the array *af*; *ldaf*≥ max(1,*n*).

ipiv — INTEGER.

Array, DIMENSION*n*. Details of the interchanges and the block structure of D as determined by ?hetrf.

colequ — LOGICAL. If *colequ* = .TRUE., column equilibration was done to *A* before calling this routine. This is needed to compute the solution and error bounds correctly.

c — REAL for cla_herfsx_extended

DOUBLE PRECISION for zla_herfsx_extended.

c contains the column scale factors for *A*. If *colequ* = .FALSE., *c* is not used.

If *c* is input, each element of *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb — INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

ldy — INTEGER. The leading dimension of the array *y*; *ldy*≥ max(1, *n*).

n_norms — INTEGER. Determines which error bounds to return. See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

If *n_norms*≥ 1, returns normwise error bounds.

If *n_norms*≥ 2, returns componentwise error bounds.

err_bnds_norm

REAL for cla_herfsx_extended

DOUBLE PRECISION for zla_herfsx_extended.

Array, DIMENSION(*nrhs*,*n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error.

Normwise relative error in the *i*-th solution vector is defined as follows:

$$\frac{\max_j \left|Xtrue_{ji} - X_{ji}\right|}{\max_j \left|X_{ji}\right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1 — "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for cla_herfsx_extended and *sqrt*(*n*)**dlamch*(ε) for zla_herfsx_extended.

err=2 — "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for cla_herfsx_extended and *sqrt*(*n*)**dlamch*(ε) for zla_herfsx_extended. This error bound should only be trusted if the previous boolean is true.

err=3 — Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for cla_herfsx_extended and *sqrt*(*n*)**dlamch*(ε) for zla_herfsx_extended to determine if the error estimate is "guaranteed". These reciprocal condition numbers are 1/(norm(1/z,inf)*norm(z,inf)) for some appropriately scaled matrix *Z*.

Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

Use this subroutine to set only the second field above.

err_bnds_comp

`REAL` for `cla_herfsx_extended`

`DOUBLE PRECISION` for `zla_herfsx_extended`.

Array, `DIMENSION(`*`nrhs`*`,`*`n_err_bnds`*`)`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is nit requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the follwoing three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for `cla_herfsx_extended` and *sqrt*(*n*)**dlamch*(ε) for `zla_herfsx_extended`.

err=2

"Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for `cla_herfsx_extended` and *sqrt*(*n*)**dlamch*(ε) for `zla_herfsx_extended`. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for

cla_herfsx_extended and *sqrt*(*n*)**dlamch*(ε) for zla_herfsx_extended to determine if the error estimate is "guaranteed". These reciprocal condition numbers are 1/(norm(1/z,inf)*norm(z,inf)) for some appropriately scaled matrix *Z*.

Let *z*=*s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1.

Use this subroutine to set only the second field above.

res, *dy*, *y_tail*

COMPLEX for cla_herfsx_extended

DOUBLE COMPLEX for zla_herfsx_extended.

Workspace arrays of DIMENSION*n*.

res holds the intermediate residual.

dy holds the intermediate solution.

y_tail holds the trailing bits of the intermediate solution.

ayb

REAL for cla_herfsx_extended

DOUBLE PRECISION for zla_herfsx_extended.

Workspace array, DIMENSION*n*.

rcond

REAL for cla_herfsx_extended

DOUBLE PRECISION for zla_herfsx_extended.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

rthresh

REAL for cla_herfsx_extended

DOUBLE PRECISION for zla_herfsx_extended.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

norm(dx_{*i*+1}) < *rthresh* * norm(dx_*i*)

where norm(*z*) is the infinity norm of *Z*.

rthresh satisfies

`0 < rthresh≤ 1`.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

`REAL` for `cla_herfsx_extended`

`DOUBLE PRECISION` for `zla_herfsx_extended`.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution *y* is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

`LOGICAL`

If `.TRUE.`, the function ignores componentwise convergence. Default value is `.FALSE.`

Output Parameters

y

`COMPLEX` for `cla_herfsx_extended`

`DOUBLE COMPLEX` for `zla_herfsx_extended`.

The improved solution matrix *Y*.

berr_out

`REAL` for `cla_herfsx_extended`

`DOUBLE PRECISION` for `zla_herfsx_extended`.

Array, `DIMENSION`*nrhs*. *berr_out(j)* contains the componentwise relative backward error for right-hand-side *j* from the formula

`max(i) ( abs(res(i)) / ( abs(op(A))*abs(y) + abs(B) )(i) )`

where `abs(z)` is the componentwise absolute value of the matrix or vector *Z*. This is computed by `?la_lin_berr`.

err_bnds_norm, *err_bnds_comp*

Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array `( 1:nrhs, 2 )`. The other elements are kept unchanged.

info

`INTEGER`. If *info* `= 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* `= -i`, the *i*-th parameter had an illegal value.

See Also

?herfsx
?hetrf
?hetrs
?lamch
ilaprec
ilatrans
?la_lin_berr

?la_herpvgrw

Computes the reciprocal pivot growth factor `norm(A)/ norm(U)` *for a Hermitian indefinite matrix.*

Syntax

```
call cla_herpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call zla_herpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
```

Include Files

- `mkl.fi`

Description

The `?la_herpvgrw` routine computes the reciprocal pivot growth factor `norm(A)/norm(U)`. The *max absolute element* norm is used. If this is much less than 1, the stability of the *LU* factorization of the equilibrated matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Specifies the triangle of A to store: If *uplo* = `'U'`, the upper triangle of *A* is stored, If *uplo* = `'L'`, the lower triangle of *A* is stored.
n	`INTEGER`. The number of linear equations, the order of the matrix *A*; *n*≥ 0.
info	`INTEGER`. The value of INFO returned from `?hetrf`, that is, the pivot in column *info* is exactly 0.
a, *af*	`COMPLEX` for `cla_herpvgrw` `DOUBLE COMPLEX` for `zla_herpvgrw`. Arrays: *a*(*lda*,*), *af*(*ldaf*,*). *a* contains the *n*-by-*n* matrix *A*. The second dimension of *a* must be at least `max(1,n)`. *af* contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by `?hetrf`. The second dimension of *af* must be at least `max(1,n)`.
lda	`INTEGER`. The leading dimension of array *a*; *lda*≥`max(1,n)`.
ldaf	`INTEGER`. The leading dimension of array *af*; *ldaf*≥`max(1,n)`.
ipiv	`INTEGER`. Array, `DIMENSION`*n*. Details of the interchanges and the block structure of D as determined by `?hetrf`.
work	`REAL` for `cla_herpvgrw` `DOUBLE PRECISION` for `zla_herpvgrw`.

Array, DIMENSION 2*n. Workspace.

See Also

?hetrf

?la_lin_berr

Computes component-wise relative backward error.

Syntax

```
call sla_lin_berr(n, nz, nrhs, res, ayb, berr )
call dla_lin_berr(n, nz, nrhs, res, ayb, berr )
call cla_lin_berr(n, nz, nrhs, res, ayb, berr )
call zla_lin_berr(n, nz, nrhs, res, ayb, berr )
```

Include Files

- mkl.fi

Description

The ?la_lin_berr computes a component-wise relative backward error from the formula:

max(*i*) (abs(R(*i*))/(abs(op(*A_s*))*abs(*Y*) + abs(*B_s*))(*i*))

where abs(*Z*) is the component-wise value of the matrix or vector *Z*.

Input Parameters

n

INTEGER. The number of linear equations, the order of the matrix *A*; *n*≥ 0.

nz

INTEGER. The parameter for guarding against spuriously zero residuals.

(*nz*+1)*slamch('Safe minimum') is added to R(*i*) in the numerator of the relative backward error formula. The default value is *n*.

nrhs

INTEGER. Number of right-hand sides, the number of columns in the matrices *AYB*, *RES*, and *BERR*; *nrhs*≥ 0.

res, *ayb*

REAL for sla_lin_berr, cla_lin_berr

DOUBLE PRECISION for dla_lin_berr, zla_lin_berr

Arrays, DIMENSION (*n*,*nrhs*).

res is the residual matrix, that is, the matrix *R* in the relative backward error formula.

ayb is the denominator of that formula, that is, the matrix abs(op(*A_s*))*abs(*Y*) + abs(*B_s*). The matrices *A*, *Y*, and *B* are from iterative refinement. See description of ?la_gerfsx_extended.

Output Parameters

berr

REAL for sla_lin_berr

DOUBLE PRECISION for dla_lin_berr

COMPLEX for cla_lin_berr

DOUBLE COMPLEX for zla_lin_berr

The component-wise relative backward error.

See Also

?lamch

?la_gerfsx_extended

?la_porcond

Estimates the Skeel condition number for a symmetric positive-definite matrix.

Syntax

```
call sla_porcond( uplo, n, a, lda, af, ldaf, cmode, c, info, work, iwork )
call dla_porcond( uplo, n, a, lda, af, ldaf, cmode, c, info, work, iwork )
```

Include Files

- mkl.fi

Description

The function estimates the Skeel condition number of

op(*A*) * op2(*C*)

where

the *cmode* parameter determines op2 as follows:

cmode **Value**	**op2(C)**
1	*C*
0	*I*
-1	inv(*C*)

The Skeel condition number

cond(*A*) = norminf(|inv(*A*)||*A*|)

is computed by computing scaling factors *R* such that

diag(*R*)*A*op2(*C*)

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

Specifies the triangle of A to store:

If *uplo* = 'U', the upper triangle of *A* is stored,

If *uplo* = 'L', the lower triangle of *A* is stored.

n INTEGER. The number of linear equations, that is, the order of the matrix *A*; *n*≥ 0.

a, *af*, *c*, *work*	REAL for sla_porcond DOUBLE PRECISION for dla_porcond Arrays: *a* (*lda*,*) contains the *n*-by-*n* matrix *A*. *af* (*ldaf*,*) contains the triangular factor L or U from the Cholesky factorization A = U^T*U or A = L*L^T, as computed by ?potrf. *c*, DIMENSION*n*. The vector *C* in the formula op(*A*) * op2(*C*). *work* is a workspace array of DIMENSION (3**n*). The second dimension of *a* and *af* must be at least max(1, *n*).
lda	INTEGER. The leading dimension of the array *ab*. *lda*≥max(1,*n*).
ldaf	INTEGER. The leading dimension of *af*. *ldaf*≥max(1,*n*).
cmode	INTEGER. Determines op2(*C*) in the formula op(*A*) * op2(*C*) as follows: If *cmode* = 1, op2(*C*) = *C*. If *cmode* = 0, op2(*C*) = *I*. If *cmode* = -1, op2(*C*) = inv(*C*).
iwork	INTEGER. Workspace array with DIMENSION*n*.

Output Parameters

info	INTEGER. If *info* = 0, the execution is successful. If *i* > 0, the *i*-th parameter is invalid.

See Also

?potrf

?la_porcond_c

*Computes the infinity norm condition number of op(A)*inv(diag(c)) for Hermitian positive-definite matrices.*

Syntax

```
call cla_porcond_c( uplo, n, a, lda, af, ldaf, c, capply, info, work, rwork )
call zla_porcond_c( uplo, n, a, lda, af, ldaf, c, capply, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

```
op(A) * inv(diag(c))
```

where the *c* is a REAL vector for cla_porcond_c and a DOUBLE PRECISION vector for zla_porcond_c.

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Specifies the triangle of A to store:
If *uplo* = 'U', the upper triangle of *A* is stored,
If *uplo* = 'L', the lower triangle of *A* is stored.

n
INTEGER. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

a
COMPLEX for cla_porcond_c
DOUBLE COMPLEX for zla_porcond_c
Array, DIMENSION(*lda*, *). On entry, the *n*-by-*n* matrix A. The second dimension of *a* must be at least max(1,*n*).

lda
INTEGER. The leading dimension of the array *a*. $lda \geq \max(1,n)$.

af
COMPLEX for cla_porcond_c
DOUBLE COMPLEX for zla_porcond_c
Array, DIMENSION(*ldaf*, *). The triangular factor L or U from the Cholesky factorization
$A = U^H*U$ or $A = L*L^H$,
as computed by ?potrf.
The second dimension of *af* must be at least max(1,*n*).

ldaf
INTEGER. The leading dimension of the array *af*. $ldaf \geq \max(1,n)$.

c
REAL for cla_porcond_c
DOUBLE PRECISION for zla_porcond_c
Array *c* with DIMENSION*n*. The vector *c* in the formula
op(*A*) * inv(diag(*c*)).

capply
LOGICAL. If .TRUE., then the function uses the vector *c* from the formula
op(*A*) * inv(diag(*c*)).

work
COMPLEX for cla_porcond_c
DOUBLE COMPLEX for zla_porcond_c
Array DIMENSION 2**n*. Workspace.

rwork
REAL for cla_porcond_c
DOUBLE PRECISION for zla_porcond_c
Array DIMENSION*n*. Workspace.

Output Parameters

info	INTEGER. If *info* = 0, the execution is successful. If *i* > 0, the *i*-th parameter is invalid.

See Also

?potrf

?la_porcond_x

*Computes the infinity norm condition number of op(A)*diag(x) for Hermitian positive-definite matrices.*

Syntax

```
call cla_porcond_x( uplo, n, a, lda, af, ldaf, x, info, work, rwork )
call zla_porcond_x( uplo, n, a, lda, af, ldaf, x, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

op(*A*) * diag(*x*)

where the *x* is a COMPLEX vector for cla_porcond_x and a DOUBLE COMPLEX vector for zla_porcond_x.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If *uplo* = 'U', the upper triangle of *A* is stored, If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The number of linear equations, that is, the order of the matrix *A*; *n*≥ 0.
a	COMPLEX for cla_porcond_c DOUBLE COMPLEX for zla_porcond_c Array, DIMENSION(*lda*, *). On entry, the *n*-by-*n* matrix A. The second dimension of *a* must be at least max(1,*n*).
lda	INTEGER. The leading dimension of the array *a*. *lda*≥max(1,*n*).
af	COMPLEX for cla_porcond_c DOUBLE COMPLEX for zla_porcond_c Array, DIMENSION(*ldaf*, *). The triangular factor L or U from the Cholesky factorization A = U^H*U or A = L*L^H,

as computed by ?potrf.

The second dimension of *af* must be at least max(1,*n*).

ldaf INTEGER. The leading dimension of the array *af*. *ldaf*≥max(1,*n*).

x COMPLEX for cla_porcond_c

DOUBLE COMPLEX for zla_porcond_c

Array *x* with DIMENSION*n*. The vector *x* in the formula

op(*A*) * inv(diag(*x*)).

work COMPLEX for cla_porcond_c

DOUBLE COMPLEX for zla_porcond_c

Array DIMENSION 2**n*. Workspace.

rwork REAL for cla_porcond_c

DOUBLE PRECISION for zla_porcond_c

Array DIMENSION*n*. Workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?potrf

?la_porfsx_extended

Improves the computed solution to a system of linear equations for symmetric or Hermitian positive-definite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

call sla_porfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call dla_porfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call cla_porfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call zla_porfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

Include Files

- mkl.fi

Description

The ?la_porfsx_extended subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The ?herfsx routine calls ?la_porfsx_extended to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

Use ?la_porfsx_extended to set only the second fields of *err_bnds_norm* and *err_bnds_comp*.

Input Parameters

prec_type

INTEGER.

Specifies the intermediate precision to be used in refinement. The value is defined by ilaprec(p), where *p* is a CHARACTER and:

If *p* = 'S': Single.

If *p* = 'D': Double.

If *p* = 'I': Indigenous.

If *p* = 'X', 'E': Extra.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Specifies the triangle of A to store:

If *uplo* = 'U', the upper triangle of *A* is stored,

If *uplo* = 'L', the lower triangle of *A* is stored.

n

INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrix *B*.

a, af, b, y

REAL for sla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended

COMPLEX for cla_porfsx_extended

DOUBLE COMPLEX for zla_porfsx_extended.

Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *y*(*ldy*,*).

The array *a* contains the original *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1,*n*).

The array *af* contains the triangular factor L or U from the Cholesky factorization as computed by ?potrf:

$A = U^T*U$ or $A = L*L^T$ for real flavors,

$A = U^H*U$ or $A = L*L^H$ for complex flavors.

The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the right-hand-side of the matrix *B*. The second dimension of *b* must be at least max(1,*nrhs*).

The array *y* on entry contains the solution matrix *X* as computed by ?potrs. The second dimension of *y* must be at least max(1,*nrhs*).

lda INTEGER. The leading dimension of the array *a*; *lda*≥ max(1,*n*).

ldaf INTEGER. The leading dimension of the array *af*; *ldaf*≥ max(1,*n*).

colequ LOGICAL. If *colequ* = .TRUE., column equilibration was done to *A* before calling this routine. This is needed to compute the solution and error bounds correctly.

c REAL for sla_porfsx_extended and cla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended and zla_porfsx_extended.

c contains the column scale factors for *A*. If *colequ* = .FALSE., *c* is not used.

If *c* is input, each element of *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

ldy INTEGER. The leading dimension of the array *y*; *ldy*≥ max(1, *n*).

n_norms INTEGER. Determines which error bounds to return. See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

If *n_norms*≥ 1, returns normwise error bounds.

If *n_norms*≥ 2, returns componentwise error bounds.

err_bnds_norm REAL for sla_porfsx_extended and cla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended and zla_porfsx_extended.

Array, DIMENSION(*nrhs*,*n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error.

Normwise relative error in the *i*-th solution vector is defined as follows:

$$\frac{\max_j \left| Xtrue_{ji} - X_{ji} \right|}{\max_j \left| X_{ji} \right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for sla_porfsx_extended/cla_porfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_porfsx_extended/zla_porfsx_extended.
err=2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for sla_porfsx_extended/cla_porfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_porfsx_extended/zla_porfsx_extended. This error bound should only be trusted if the previous boolean is true.
err=3	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for sla_porfsx_extended/cla_porfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_porfsx_extended/zla_porfsx_extended to determine if the error estimate is "guaranteed". These reciprocal condition numbers are 1/(norm(1/z,inf)*norm(z,inf)) for some appropriately scaled matrix *Z*. Let *z=s*a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1. Use this subroutine to set only the second field above.

err_bnds_comp

REAL for sla_porfsx_extended and cla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended and zla_porfsx_extended.

Array, DIMENSION(*nrhs*,*n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is nit requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the follwoing three fields:

err=1 — "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for sla_porfsx_extended/cla_porfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_porfsx_extended/zla_porfsx_extended.

err=2 — "Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for sla_porfsx_extended/cla_porfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_porfsx_extended/zla_porfsx_extended. This error bound should only be trusted if the previous boolean is true.

err=3 — Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for sla_porfsx_extended/cla_porfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_porfsx_extended/zla_porfsx_extended to determine if the error estimate is "guaranteed". These reciprocal condition numbers are 1/(norm(1/z,inf)*norm(z,inf)) for some appropriately scaled matrix *Z*.

Let *z=s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1.

Use this subroutine to set only the second field above.

res, *dy*, *y_tail*

REAL for sla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended

COMPLEX for cla_porfsx_extended

DOUBLE COMPLEX for zla_porfsx_extended.

Workspace arrays of DIMENSION*n*.

res holds the intermediate residual.

dy holds the intermediate solution.

y_tail holds the trailing bits of the intermediate solution.

ayb

REAL for sla_porfsx_extended and cla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended and zla_porfsx_extended.

Workspace array, DIMENSION*n*.

rcond

REAL for sla_porfsx_extended and cla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended and zla_porfsx_extended.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

rthresh

REAL for sla_porfsx_extended and cla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended and zla_porfsx_extended.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

norm(dx_{*i*+1}) < *rthresh* * norm(dx_*i*)

where norm(*z*) is the infinity norm of *Z*.

rthresh satisfies

0 < *rthresh*≤ 1.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

REAL for sla_porfsx_extended and cla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended and zla_porfsx_extended.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution *y* is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

LOGICAL

If .TRUE., the function ignores componentwise convergence. Default value is .FALSE.

Output Parameters

y

REAL for sla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended

COMPLEX for cla_porfsx_extended

DOUBLE COMPLEX for zla_porfsx_extended.

The improved solution matrix *Y*.

berr_out

REAL for sla_porfsx_extended and cla_porfsx_extended

DOUBLE PRECISION for dla_porfsx_extended and zla_porfsx_extended.

Array, DIMENSION*nrhs*. *berr_out(j)* contains the componentwise relative backward error for right-hand-side *j* from the formula

max(*i*) (abs(*res*(*i*)) / (abs(op(*A*))*abs(*y*) + abs(*B*))(*i*))

where abs(*z*) is the componentwise absolute value of the matrix or vector *Z*. This is computed by ?la_lin_berr.

err_bnds_norm, *err_bnds_comp*

Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array (1:*nrhs*, 2). The other elements are kept unchanged.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

See Also

?porfsx
?potrf
?potrs
?lamch
ilaprec
ilatrans
?la_lin_berr

?la_porpvgrw

Computes the reciprocal pivot growth factor `norm(A)/ norm(U)` *for a symmetric or Hermitian positive-definite matrix.*

Syntax

```
call sla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
call dla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
call cla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
call zla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
```

Include Files

- `mkl.fi`

Description

The `?la_porpvgrw` routine computes the reciprocal pivot growth factor `norm(A)/norm(U)`. The *max absolute element* norm is used. If this is much less than 1, the stability of the *LU* factorization of the equilibrated matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Specifies the triangle of A to store:

If *uplo* = `'U'`, the upper triangle of *A* is stored,

If *uplo* = `'L'`, the lower triangle of *A* is stored.

ncols

`INTEGER`. The number of columns of the matrix *A*; *ncols*≥ 0.

a, *af*

`REAL` for `sla_porpvgrw`

`DOUBLE PRECISION` for `dla_porpvgrw`

`COMPLEX` for `cla_porpvgrw`

`DOUBLE COMPLEX` for `zla_porpvgrw`.

Arrays: *a*(*lda*,*), *af*(*ldaf*,*).

The array *a* contains the input *n*-by-*n* matrix *A*. The second dimension of *a* must be at least `max(1,n)`.

The array *af* contains the triangular factor `L` or `U` from the Cholesky factorization as computed by `?potrf`:

$A = U^T*U$ or $A = L*L^T$ for real flavors,

$A = U^H*U$ or $A = L*L^H$ for complex flavors.

The second dimension of *af* must be at least `max(1,n)`.

lda

`INTEGER`. The leading dimension of *a*; *lda*≥ `max(1,n)`.

ldaf

`INTEGER`. The leading dimension of *af*; *ldaf*≥ `max(1,n)`.

work REAL for sla_porpvgrw and cla_porpvgrw

DOUBLE PRECISION for dla_porpvgrw and zla_porpvgrw.

Workspace array, dimension 2*n.

See Also

?potrf

?laqhe

Scales a Hermitian matrix.

Syntax

```
call claqhe( uplo, n, a, lda, s, scond, amax, equed )
call zlaqhe( uplo, n, a, lda, s, scond, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a Hermitian matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo CHARACTER*1.

Specifies whether to store the upper or lower part of the Hermitian matrix *A*.

If *uplo* = 'U', the upper triangular part of *A*;

if *uplo* = 'L', the lower triangular part of *A*.

n INTEGER. The order of the matrix *A*.

$n \geq 0$.

a COMPLEX for claqhe

DOUBLE COMPLEX for zlaqhe

Array, DIMENSION (*lda*,*n*). On entry, the Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of matrix *A* and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of matrix *A* and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*.

$lda \geq \max(n,1)$.

s REAL for claqhe

DOUBLE PRECISION for zlaqhe

Array, DIMENSION (*n*). The scale factors for *A*.

scond

REAL for claqhe

DOUBLE PRECISION for zlaqhe

Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax

REAL for claqhe

DOUBLE PRECISION for zlaqhe

Absolute value of largest matrix entry.

Output Parameters

a

If *equed* = 'Y', *a* contains the equilibrated matrix diag(*s*)*A*diag(*s*).

equed

CHARACTER*1.

Specifies whether or not equilibration was done.

If *equed* = 'N': No equilibration.

If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(*s*)*A*diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*. The parameter *thresh* is a threshold value used to decide if scaling should be done based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done.

The *large* and *small* parameters are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqhp

Scales a Hermitian matrix stored in packed form.

Syntax

```
call claqhp( uplo, n, ap, s, scond, amax, equed )
call zlaqhp( uplo, n, ap, s, scond, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a Hermitian matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo

CHARACTER*1.

Specifies whether to store the upper or lower part of the Hermitian matrix *A*.

If *uplo* = 'U', the upper triangular part of *A*;

if *uplo* = 'L', the lower triangular part of *A*.

n

INTEGER. The order of the matrix *A*.

$n \geq 0$.

ap

COMPLEX for claqhp

DOUBLE COMPLEX for zlaqhp

Array, DIMENSION ($n*(n+1)/2$). The Hermitian matrix *A*.

- If *uplo* = 'U', the upper triangular part of the Hermitian matrix *A* is stored in the packed array *ap* as follows:

  ap(*i*+(*j*-1)**j*/2) = *A*(*i*,*j*) for $1 \leq i \leq j$.
- If *uplo* = 'L', the lower triangular part of Hermitian matrix *A* is stored in the packed array *ap* as follows:

  ap(*i*+(*j*-1)*(2*n*-j)/2) = *A*(*i*,*j*) for $j \leq i \leq n$.

s

REAL for claqhp

DOUBLE PRECISION for zlaqhp

Array, DIMENSION (*n*). The scale factors for *A*.

scond

REAL for claqhp

DOUBLE PRECISION for zlaqhp

Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax

REAL for claqhp

DOUBLE PRECISION for zlaqhp

Absolute value of largest matrix entry.

Output Parameters

a

If *equed* = 'Y', *a* contains the equilibrated matrix diag(*s*)*A*diag(*s*) in the same storage format as on input.

equed

CHARACTER*1.

Specifies whether or not equilibration was done.

If *equed* = 'N': No equilibration.

If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(*s*)*A*diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*. The parameter *thresh* is a threshold value used to decide if scaling should be done based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done.

The *large* and *small* parameters are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?larcm

Multiplies a square real matrix by a complex matrix.

Syntax

```
call clarcm( m, n, a, lda, b, ldb, c, ldc, rwork )
call zlarcm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

Description

The routine performs a simple matrix-matrix multiplication of the form

`C = A*B`,

where *A* is *m*-by-*m* and real, *B* is *m*-by-*n* and complex, and *C* is *m*-by-*n* and complex.

Input Parameters

m	INTEGER. The number of rows and columns of matrix *A* and the number of rows of matrix *C* (*m*≥ 0).
n	INTEGER. The number of columns of matrix *B* and the number of columns of matrix *C* (*n*≥ 0).
a	REAL for clarcm DOUBLE PRECISION for zlarcm Array, size (*lda*, *m*). Contains the *m*-by-*m* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*, *lda*≥max(1, *m*).
b	COMPLEX for clarcm DOUBLE COMPLEX for zlarcm Array, DIMENSION(*ldb*, *n*). Contains the *m*-by-*n* matrix *B*.
ldb	INTEGER. The leading dimension of the array *b*, *ldb*≥max(1, *m*) for column-major layout; *ldb*≥max(1, *n*) for row-major layout .
ldc	INTEGER. The leading dimension of the array *c*, *ldc*≥max(1, *m*) for column-major layout; *ldc*≥max(1, *n*) for row-major layout .
rwork	REAL for clarcm DOUBLE PRECISION for zlarcm Workspace array, DIMENSION(2*m*n).

Output Parameters

c	COMPLEX for clarcm DOUBLE COMPLEX for zlarcm Array, size (*ldc*, *n*). Contains the *m*-by-*n* matrix *C*.

Return Values

This function returns a value *info*. If *info* = 0, the execution is successful. If *info* = -*i*, parameter *i* had an illegal value.

?la_gerpvgrw

Computes the reciprocal pivot growth factor `norm(A)/ norm(U)` *for a general matrix.*

Syntax

```
call sla_gerpvgrw( n, ncols, a, lda, af, ldaf )
call dla_gerpvgrw( n, ncols, a, lda, af, ldaf )
call cla_gerpvgrw( n, ncols, a, lda, af, ldaf )
call zla_gerpvgrw( n, ncols, a, lda, af, ldaf )
```

Include Files

- `mkl.fi`

Description

The `?la_gerpvgrw` routine computes the reciprocal pivot growth factor `norm(A)/norm(U)`. The *max absolute element* norm is used. If this is much less than 1, the stability of the *LU* factorization of the equilibrated matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

n	`INTEGER`. The number of linear equations, the order of the matrix *A*; *n*≥ 0.
ncols	`INTEGER`. The number of columns of the matrix *A*; *ncols*≥ 0.

a, *af*	`REAL` for `sla_gerpvgrw`
	`DOUBLE PRECISION` for `dla_gerpvgrw`
	`COMPLEX` for `cla_gerpvgrw`
	`DOUBLE COMPLEX` for `zla_gerpvgrw`.
	Arrays: `a(lda,*)`, `af(ldaf,*)`.
	The array *a* contains the input *n*-by-*n* matrix *A*. The second dimension of *a* must be at least `max(1,n)`.
	The array *af* contains the factors L and U from the factorization triangular factor `L` or `U` from the Cholesky factorization `A = P*L*U` as computed by `?getrf`. The second dimension of *af* must be at least `max(1,n)`.

lda	`INTEGER`. The leading dimension of *a*; *lda*≥ `max(1,n)`.
ldaf	`INTEGER`. The leading dimension of *af*; *ldaf*≥ `max(1,n)`.

See Also

`?getrf`

?larscl2

Performs reciprocal diagonal scaling on a vector.

Syntax

```
call slarscl2(m, n, d, x, ldx)
call dlarscl2(m, n, d, x, ldx)
call clarscl2(m, n, d, x, ldx)
call zlarscl2(m, n, d, x, ldx)
```

Include Files

- mkl.fi

Description

The ?larscl2 routines perform reciprocal diagonal scaling on a vector

$x := D^{-1}*x$,

where:

x is a vector, and

D is a diagonal matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *D* and the number of elements of the vector *x*. The value of *m* must be at least zero.
n	INTEGER. The number of columns of *D* and *x*. The value of *n* must be at least zero.
d	REAL for slarscl2 and clarscl2. DOUBLE PRECISION for dlarscl2 and zlarscl2. Array, DIMENSION*m*. Diagonal matrix *D* stored as a vector of length *m*.
x	REAL for slarscl2. DOUBLE PRECISION for dlarscl2. COMPLEX for clarscl2. DOUBLE COMPLEX for zlarscl2. Array, DIMENSION(*ldx*,*n*). The vector *x* to scale by *D*.
ldx	INTEGER. The leading dimension of the vector *x*. The value of *ldx* must be at least zero.

Output Parameters

x	Scaled vector *x*.

?lascl2

Performs diagonal scaling on a vector.

Syntax

```
call slascl2(m, n, d, x, ldx)
call dlascl2(m, n, d, x, ldx)
call clascl2(m, n, d, x, ldx)
call zlascl2(m, n, d, x, ldx)
```

Include Files

- mkl.fi

Description

The ?lascl2 routines perform diagonal scaling on a vector

x := *D***x*,

where:

x is a vector, and

D is a diagonal matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *D* and the number of elements of the vector *x*. The value of *m* must be at least zero.
n	INTEGER. The number of columns of *D* and *x*. The value of *n* must be at least zero.
d	REAL for slascl2 and clascl2. DOUBLE PRECISION for dlascl2 and zlascl2. Array, DIMENSION*m*. Diagonal matrix *D* stored as a vector of length *m*.
x	REAL for slascl2. DOUBLE PRECISION for dlascl2. COMPLEX for clascl2. DOUBLE COMPLEX for zlascl2. Array, DIMENSION(*ldx*,*n*). The vector *x* to scale by *D*.
ldx	INTEGER. The leading dimension of the vector *x*. The value of *ldx* must be at least zero.

Output Parameters

x	Scaled vector *x*.

?la_syamv

Computes a matrix-vector product using a symmetric indefinite matrix to calculate error bounds.

Syntax

```
call sla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call dla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call cla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call zla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

Include Files

- mkl.fi

Description

The ?la_syamv routines perform a matrix-vector operation defined as

y := *alpha**abs(*A*)*abs(*x*) + *beta**abs(*y*),

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *n*-by-*n* Hermitian matrix.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by (*n* + 1) times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the array A is to be referenced:

If *uplo* = 'BLAS_UPPER', only the upper triangular part of A is to be referenced,

If *uplo* = 'BLAS_LOWER', only the lower triangular part of A is to be referenced.

n

INTEGER. Specifies the number of rows and columns of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for sla_syamv and cla_syamv

DOUBLE PRECISION for dla_syamv and zla_syamv.

Specifies the scalar *alpha*.

a

REAL for sla_syamv

DOUBLE PRECISION for dla_syamv

COMPLEX for cla_syamv

	DOUBLE COMPLEX for zla_syamv. Array, DIMENSION(*lda*, *). Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients. The second dimension of *a* must be at least max(1,*n*).
lda	INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).
x	REAL for sla_syamv DOUBLE PRECISION for dla_syamv COMPLEX for cla_syamv DOUBLE COMPLEX for zla_syamv. Array, DIMENSION at least (1+(*n*–1)*abs(*incx*)). Before entry, the incremented array *x* must contain the vector *X*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must be non-zero.
beta	REAL for sla_syamv and cla_syamv DOUBLE PRECISION for dla_syamv and zla_syamv Specifies the scalar *beta*. When *beta* is zero, you do not need to set *y* on input.
y	REAL for sla_syamv and cla_syamv DOUBLE PRECISION for dla_syamv and zla_syamv Array, DIMENSION at least (1 +(*n* – 1)*abs(*incy*)) otherwise. Before entry with non-zero *beta*, the incremented array *y* must contain the vector *Y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must be non-zero.

Output Parameters

y	Updated vector *Y*.

?la_syrcond

Estimates the Skeel condition number for a symmetric indefinite matrix.

Syntax

```
call sla_syrcond( uplo, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
call dla_syrcond( uplo, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
```

Include Files

- mkl.fi

Description

The function estimates the Skeel condition number of

`op(A) * op2(C)`

where

the *cmode* parameter determines `op2` as follows:

cmode **Value**	**op2(C)**
1	*C*
0	*I*
-1	`inv(C)`

The Skeel condition number

`cond(A) = norminf(|inv(A)||A|)`

is computed by computing scaling factors *R* such that

`diag(R)*A*op2(C)`

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

uplo

`CHARACTER*1`. Must be `'U'` or `'L'`.

Specifies the triangle of A to store:

If *uplo* = `'U'`, the upper triangle of *A* is stored,

If *uplo* = `'L'`, the lower triangle of *A* is stored.

n

`INTEGER`. The number of linear equations, that is, the order of the matrix *A*; *n*≥ 0.

a, *af*, *c*, *work*

`REAL` for `sla_syrcond`

`DOUBLE PRECISION` for `dla_syrcond`

Arrays:

ab (*lda*,*) contains the *n*-by-*n* matrix *A*.

af (*ldaf*,*) contains the The block diagonal matrix D and the multipliers used to obtain the factor `L` or `U` as computed by `?sytrf`.

The second dimension of *a* and *af* must be at least `max(1, n)`.

c, `DIMENSION`*n*. The vector *C* in the formula `op(A) * op2(C)`.

work is a workspace array of `DIMENSION` (3**n*).

lda

`INTEGER`. The leading dimension of the array *ab*. *lda*≥`max(1,n)`.

ldaf

`INTEGER`. The leading dimension of *af*. *ldaf*≥`max(1,n)`.

ipiv

`INTEGER`.

Array with `DIMENSION`*n*. Details of the interchanges and the block structure of D as determined by `?sytrf`.

cmode INTEGER. Determines op2(*C*) in the formula op(*A*) * op2(*C*) as follows:

If *cmode* = 1, op2(*C*) = *C*.

If *cmode* = 0, op2(*C*) = *I*.

If *cmode* = -1, op2(*C*) = inv(*C*).

iwork INTEGER. Workspace array with DIMENSION*n*.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?sytrf

?la_syrcond_c

*Computes the infinity norm condition number of op(A)*inv(diag(c)) for symmetric indefinite matrices.*

Syntax

```
call cla_syrcond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
call zla_syrcond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

op(*A*) * inv(diag(c))

where the *c* is a REAL vector for cla_syrcond_c and a DOUBLE PRECISION vector for zla_syrcond_c.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

Specifies the triangle of A to store:

If *uplo* = 'U', the upper triangle of *A* is stored,

If *uplo* = 'L', the lower triangle of *A* is stored.

n INTEGER. The number of linear equations, that is, the order of the matrix *A*; *n*≥ 0.

a COMPLEX for cla_syrcond_c

DOUBLE COMPLEX for zla_syrcond_c

Array, DIMENSION(*lda*, *). On entry, the *n*-by-*n* matrix A. The second dimension of *a* must be at least max(1,*n*).

lda INTEGER. The leading dimension of the array *a*. *lda*≥max(1,*n*).

af COMPLEX for cla_syrcond_c

DOUBLE COMPLEX for zla_syrcond_c

Array, DIMENSION(*ldaf*, *). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf. The second dimension of *af* must be at least max(1,*n*).

ldaf INTEGER. The leading dimension of the array *af*. *ldaf*≥max(1,*n*).

ipiv INTEGER.

Array with DIMENSION*n*. Details of the interchanges and the block structure of D as determined by ?sytrf.

c REAL for cla_syrcond_c

DOUBLE PRECISION for zla_syrcond_c

Array *c* with DIMENSION*n*. The vector *c* in the formula

op(*A*) * inv(diag(*c*)).

capply LOGICAL. If .TRUE., then the function uses the vector *c* from the formula

op(*A*) * inv(diag(*c*)).

work COMPLEX for cla_syrcond_c

DOUBLE COMPLEX for zla_syrcond_c

Array DIMENSION 2**n*. Workspace.

rwork REAL for cla_syrcond_c

DOUBLE PRECISION for zla_syrcond_c

Array DIMENSION*n*. Workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?sytrf

?la_syrcond_x

*Computes the infinity norm condition number of op(A)*diag(x) for symmetric indefinite matrices.*

Syntax

```
call cla_syrcond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
call zla_syrcond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

op(*A*) * diag(*x*)

where the *x* is a COMPLEX vector for cla_syrcond_x and a DOUBLE COMPLEX vector for zla_syrcond_x.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Specifies the triangle of A to store:

If *uplo* = 'U', the upper triangle of *A* is stored,

If *uplo* = 'L', the lower triangle of *A* is stored.

n

INTEGER. The number of linear equations, that is, the order of the matrix *A*; *n*≥ 0.

a

COMPLEX for cla_syrcond_c

DOUBLE COMPLEX for zla_syrcond_c

Array, DIMENSION(*lda*, *). On entry, the *n*-by-*n* matrix A. The second dimension of *a* must be at least max(1,*n*).

lda

INTEGER. The leading dimension of the array *a*. *lda*≥max(1,*n*).

af

COMPLEX for cla_syrcond_c

DOUBLE COMPLEX for zla_syrcond_c

Array, DIMENSION(*ldaf*, *). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf. The second dimension of *af* must be at least max(1,*n*).

ldaf

INTEGER. The leading dimension of the array *af*. *ldaf*≥max(1,*n*).

ipiv

INTEGER.

Array with DIMENSION*n*. Details of the interchanges and the block structure of D as determined by ?sytrf.

x

COMPLEX for cla_syrcond_c

DOUBLE COMPLEX for zla_syrcond_c

Array *x* with DIMENSION*n*. The vector *x* in the formula

op(*A*) * inv(diag(*x*)).

work

COMPLEX for cla_syrcond_c

DOUBLE COMPLEX for zla_syrcond_c

Array DIMENSION 2**n*. Workspace.

rwork

REAL for cla_syrcond_c

DOUBLE PRECISION for zla_syrcond_c

Array DIMENSION*n*. Workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

?sytrf

?la_syrfsx_extended

Improves the computed solution to a system of linear equations for symmetric indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

call sla_syrfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call dla_syrfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call cla_syrfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

call zla_syrfsx_extended(*prec_type*, *uplo*, *n*, *nrhs*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *colequ*, *c*, *b*, *ldb*, *y*, *ldy*, *berr_out*, *n_norms*, *err_bnds_norm*, *err_bnds_comp*, *res*, *ayb*, *dy*, *y_tail*, *rcond*, *ithresh*, *rthresh*, *dz_ub*, *ignore_cwise*, *info*)

Include Files

- mkl.fi

Description

The ?la_syrfsx_extended subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The ?syrfsx routine calls ?la_syrfsx_extended to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

Use ?la_syrfsx_extended to set only the second fields of *err_bnds_norm* and *err_bnds_comp*.

Input Parameters

prec_type INTEGER.

Specifies the intermediate precision to be used in refinement. The value is defined by ilaprec(p), where *p* is a CHARACTER and:

If *p* = 'S': Single.

If *p* = 'D': Double.

If *p* = 'I': Indigenous.

If *p* = 'X', 'E': Extra.

uplo — CHARACTER*1. Must be 'U' or 'L'.

Specifies the triangle of A to store:

If *uplo* = 'U', the upper triangle of *A* is stored,

If *uplo* = 'L', the lower triangle of *A* is stored.

n — INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides; the number of columns of the matrix *B*.

a, af, b, y — REAL for sla_syrfsx_extended

DOUBLE PRECISION for dla_syrfsx_extended

COMPLEX for cla_syrfsx_extended

DOUBLE COMPLEX for zla_syrfsx_extended.

Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *y*(*ldy*,*).

The array *a* contains the original *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1,*n*).

The array *af* contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf.

The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the right-hand-side of the matrix *B*. The second dimension of *b* must be at least max(1,*nrhs*).

The array *y* on entry contains the solution matrix *X* as computed by ?sytrs. The second dimension of *y* must be at least max(1,*nrhs*).

lda — INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1,*n*).

ldaf — INTEGER. The leading dimension of the array *af*; *ldaf* ≥ max(1,*n*).

ipiv — INTEGER.

Array with DIMENSION*n*. Details of the interchanges and the block structure of D as determined by ?sytrf.

colequ — LOGICAL. If *colequ* = .TRUE., column equilibration was done to *A* before calling this routine. This is needed to compute the solution and error bounds correctly.

c — REAL for sla_syrfsx_extended and cla_syrfsx_extended

DOUBLE PRECISION for dla_syrfsx_extended and zla_syrfsx_extended.

c contains the column scale factors for *A*. If *colequ* = .FALSE., *c* is not used.

If *c* is input, each element of *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb — INTEGER. The leading dimension of the array *b*; *ldb*≥ max(1, *n*).

ldy — INTEGER. The leading dimension of the array *y*; *ldy*≥ max(1, *n*).

n_norms — INTEGER. Determines which error bounds to return. See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

If *n_norms*≥ 1, returns normwise error bounds.

If *n_norms*≥ 2, returns componentwise error bounds.

err_bnds_norm — REAL for sla_syrfsx_extended and cla_syrfsx_extended

DOUBLE PRECISION for dla_syrfsx_extended and zla_syrfsx_extended.

Array, DIMENSION(*nrhs*,*n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error.

Normwise relative error in the *i*-th solution vector is defined as follows:

$$\frac{\max_j \left| Xtrue_{ji} - X_{ji} \right|}{\max_j \left| X_{ji} \right|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1 — "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for sla_syrfsx_extended/cla_syrfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_syrfsx_extended/zla_syrfsx_extended.

err=2 — "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for sla_syrfsx_extended/cla_syrfsx_extended and *sqrt*(*n*)**dlamch*(ε) for

`dla_syrfsx_extended/zla_syrfsx_extended`. This error bound should only be trusted if the previous boolean is true.

err=3 — Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for `sla_syrfsx_extended/cla_syrfsx_extended` and *sqrt*(*n*)**dlamch*(ε) for `dla_syrfsx_extended/zla_syrfsx_extended` to determine if the error estimate is "guaranteed". These reciprocal condition numbers are `1/(norm(1/z,inf)*norm(z,inf))` for some appropriately scaled matrix *Z*.

Let *z*=*s***a*, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

Use this subroutine to set only the second field above.

err_bnds_comp — `REAL` for `sla_syrfsx_extended` and `cla_syrfsx_extended`

`DOUBLE PRECISION` for `dla_syrfsx_extended` and `zla_syrfsx_extended`.

Array, `DIMENSION(`*nrhs*`,`*n_err_bnds*`)`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{\left|Xtrue_{ji} - X_{ji}\right|}{\left|X_{ji}\right|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is nit requested (*params(3)* = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *err_bnds_comp(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:,err)* contains the follwoing three fields:

err=1 — "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ε) for

		sla_syrfsx_extended/cla_syrfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_syrfsx_extended/zla_syrfsx_extended.
	err=2	"Guaranteed" error bpound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ε) for sla_syrfsx_extended/cla_syrfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_syrfsx_extended/zla_syrfsx_extended. This error bound should only be trusted if the previous boolean is true.
	err=3	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ε) for sla_syrfsx_extended/cla_syrfsx_extended and *sqrt*(*n*)**dlamch*(ε) for dla_syrfsx_extended/zla_syrfsx_extended to determine if the error estimate is "guaranteed". These reciprocal condition numbers are 1/(norm(1/z,inf)*norm(z,inf)) for some appropriately scaled matrix *Z*. Let *z*=*s**(*a**diag(*x*)), where *x* is the solution for the current right-hand side and *s* scales each row of *a**diag(*x*) by a power of the radix so all absolute row sums of *z* are approximately 1. Use this subroutine to set only the second field above.
res, *dy*, *y_tail*		REAL for sla_syrfsx_extended DOUBLE PRECISION for dla_syrfsx_extended COMPLEX for cla_syrfsx_extended DOUBLE COMPLEX for zla_syrfsx_extended. Workspace arrays of DIMENSION*n*. *res* holds the intermediate residual. *dy* holds the intermediate solution. *y_tail* holds the trailing bits of the intermediate solution.
ayb		REAL for sla_syrfsx_extended and cla_syrfsx_extended DOUBLE PRECISION for dla_syrfsx_extended and zla_syrfsx_extended. Workspace array, DIMENSION*n*.

rcond

REAL for sla_syrfsx_extended and cla_syrfsx_extended

DOUBLE PRECISION for dla_syrfsx_extended and zla_syrfsx_extended.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

rthresh

REAL for sla_syrfsx_extended and cla_syrfsx_extended

DOUBLE PRECISION for dla_syrfsx_extended and zla_syrfsx_extended.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

norm(dx_{*i*+1}) < *rthresh* * norm(dx_*i*)

where norm(*z*) is the infinity norm of *Z*.

rthresh satisfies

0 < *rthresh*≤ 1.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

REAL for sla_syrfsx_extended and cla_syrfsx_extended

DOUBLE PRECISION for dla_syrfsx_extended and zla_syrfsx_extended.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution *y* is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

LOGICAL

If .TRUE., the function ignores componentwise convergence. Default value is .FALSE.

Output Parameters

y

REAL for sla_syrfsx_extended

DOUBLE PRECISION for dla_syrfsx_extended

COMPLEX for cla_syrfsx_extended

DOUBLE COMPLEX for zla_syrfsx_extended.

	The improved solution matrix *Y*.
berr_out	REAL for sla_syrfsx_extended and cla_syrfsx_extended DOUBLE PRECISION for dla_syrfsx_extended and zla_syrfsx_extended. Array, DIMENSION*nrhs*. *berr_out(j)* contains the componentwise relative backward error for right-hand-side *j* from the formula max(*i*) (abs(*res*(*i*)) / (abs(op(*A*))*abs(*y*) + abs(*B*))(*i*)) where abs(*z*) is the componentwise absolute value of the matrix or vector *Z*. This is computed by ?la_lin_berr.
err_bnds_norm, *err_bnds_comp*	Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array (1:*nrhs*, 2). The other elements are kept unchanged.
info	INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed. If *info* = -*i*, the *i*-th parameter had an illegal value.

See Also

?syrfsx
?sytrf
?sytrs
?lamch
ilaprec
ilatrans
?la_lin_berr

?la_syrpvgrw

Computes the reciprocal pivot growth factor norm(A)/ norm(U) for a symmetric indefinite matrix.

Syntax

```
call sla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call dla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call cla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call zla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
```

Include Files

- mkl.fi

Description

The ?la_syrpvgrw routine computes the reciprocal pivot growth factor norm(*A*)/norm(*U*). The *max absolute element* norm is used. If this is much less than 1, the stability of the *LU* factorization of the equilibrated matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If *uplo* = 'U', the upper triangle of *A* is stored, If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The number of linear equations, the order of the matrix *A*; *n*≥ 0.
info	INTEGER. The value of INFO returned from ?sytrf, that is, the pivot in column *info* is exactly 0.
a, *af*	REAL for sla_syrpvgrw DOUBLE PRECISION for dla_syrpvgrw COMPLEX for cla_syrpvgrw DOUBLE COMPLEX for zla_syrpvgrw. Arrays: *a*(*lda*,*), *af*(*ldaf*,*). The array *a* contains the input *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1,*n*). The array *af* contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf. The second dimension of *af* must be at least max(1,*n*).
lda	INTEGER. The leading dimension of *a*; *lda*≥ max(1,*n*).
ldaf	INTEGER. The leading dimension of *af*; *ldaf*≥ max(1,*n*).
ipiv	INTEGER. Array, DIMENSION*n*. Details of the interchanges and the block structure of D as determined by ?sytrf.
work	REAL for sla_syrpvgrw and cla_syrpvgrw DOUBLE PRECISION for dla_syrpvgrw and zla_syrpvgrw. Workspace array, dimension 2**n*.

See Also

?sytrf

?la_wwaddw

Adds a vector into a doubled-single vector.

Syntax

```
call sla_wwaddw( n, x, y, w )
call dla_wwaddw( n, x, y, w )
call cla_wwaddw( n, x, y, w )
call zla_wwaddw( n, x, y, w )
```

Include Files

- mkl.fi

Description

The ?la_wwaddw routine adds a vector *W* into a doubled-single vector (*X, Y*). This works for all existing IBM hex and binary floating-point arithmetics, but not for decimal.

Input Parameters

n	INTEGER. The length of vectors *X, Y*, and *W* .
x, *y*, *w*	REAL for sla_wwaddw DOUBLE PRECISION for dla_wwaddw COMPLEX for cla_wwaddw DOUBLE COMPLEX for zla_wwaddw. Arrays DIMENSION*n*. *x* and *y* contain the first and second parts of the doubled-single accumulation vector, respectively. *w* contains the vector *W* to be added.

Output Parameters

x, *y*	Contain the first and second parts of the doubled-single accumulation vector, respectively, after adding the vector *W*.

mkl_?tppack

Copies a triangular/symmetric matrix or submatrix from standard full format to standard packed format.

Syntax

```
call mkl_stppack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_dtppack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_ctppack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_ztppack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_tppack (ap, i, j, rows, cols, a[, uplo] [, trans] [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine copies a triangular or symmetric matrix or its submatrix from standard full format to packed format

$AP_{i:i+rows-1,\ j:j+cols-1}$:= op(*A*)

Standard packed formats include:

- TP: triangular packed storage

- SP: symmetric indefinite packed storage
- HP: Hermitian indefinite packed storage
- PP: symmetric or Hermitian positive definite packed storage

Full formats include:

- GE: general
- TR: triangular
- SY: symmetric indefinite
- HE: Hermitian indefinite
- PO: symmetric or Hermitian positive definite

NOTE
Any elements of the copied submatrix rectangular outside of the triangular part of the matrix *AP* are skipped.

Input Parameters

The data types are given for the Fortran interface.

uplo	CHARACTER*1. Specifies whether the matrix *AP* is upper or lower triangular. If *uplo* = 'U', *AP* is upper triangular. If *uplo* = 'L': *AP* is lower triangular.
trans	CHARACTER*1. Specifies whether or not the copied block of *A* is transposed or not. If *trans* = 'N', no transpose: op(*A*) = *A*. If *trans* = 'T',transpose: op(*A*) = A^T. If *trans* = 'C',conjugate transpose: op(*A*) = A^H. For real data this is the same as *trans* = 'T'.
n	INTEGER. The order of the matrix *AP*; $n \geq 0$
i, j	INTEGER. Coordinates of the left upper corner of the destination submatrix in *AP*. If *uplo*='U', $1 \leq i \leq j \leq n$. If *uplo*='L', $1 \leq j \leq i \leq n$.
rows	INTEGER. Number of rows in the destination submatrix. $0 \leq rows \leq n - i + 1$.
cols	INTEGER. Number of columns in the destination submatrix. $0 \leq cols \leq n - j + 1$.
a	REAL for mkl_stppack DOUBLE PRECISION for mkl_dtppack COMPLEX for mkl_ctppack DOUBLE COMPLEX for mkl_ztppack Pointer to the source submatrix.

Array *a*(*lda*, *) contains the *rows*-by-*cols* submatrix stored as unpacked rows-by-columns if *trans* = 'N', or unpacked columns-by-rows if *trans* = 'T' or *trans* = 'C'. The size of *a* must be at least *lda***cols* for *trans* = 'N' or *lda***rows* for *trans*='T' or *trans*='C'.

NOTE
If there are elements outside of the triangular part of *AP*, they are skipped and are not copied from *a*.

lda INTEGER. The leading dimension of the array *a*.

lda≥ max(1, *rows*) for *trans* = 'N' and and *lda*≥ max(1, *cols*) for *trans*='T' or *trans*='C'.

Output Parameters

ap

`REAL` for `mkl_stppack`

`DOUBLE PRECISION` for `mkl_dtppack`

`COMPLEX` for `mkl_ctppack`

`DOUBLE COMPLEX` for `mkl_ztppack`

Array of size at least max(1, *n*(*n*+1)/2). The array *ap* contains either the upper or the lower triangular part of the matrix *AP* (as specified by *uplo*) in packed storage (see Matrix Storage Schemes). The submatrix of *ap* from row *i* to row *i* + *rows* - 1 and column *j* to column *j* + *cols* - 1 is overwritten with a copy of the source matrix.

info INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

mkl_?tpunpack

Copies a triangular/symmetric matrix or submatrix from standard packed format to full format.

Syntax

```
call mkl_stpunpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_dtpunpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_ctpunpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_ztpunpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_tpunpack (ap, i, j, rows, cols, a[, uplo] [, trans] [, info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine copies a triangular or symmetric matrix or its submatrix from standard packed format to full format.

$A := \mathrm{op}(AP_{i:i+rows-1,\ j:j+cols-1})$

Standard packed formats include:

- TP: triangular packed storage
- SP: symmetric indefinite packed storage
- HP: Hermitian indefinite packed storage
- PP: symmetric or Hermitian positive definite packed storage

Full formats include:

- GE: general
- TR: triangular
- SY: symmetric indefinite
- HE: Hermitian indefinite
- PO: symmetric or Hermitian positive definite

NOTE
Any elements of the copied submatrix rectangular outside of the triangular part of *AP* are skipped.

Input Parameters

The data types are given for the Fortran interface.

uplo	CHARACTER*1. Specifies whether matrix *AP* is upper or lower triangular. If *uplo* = 'U', *AP* is upper triangular. If *uplo* = 'L': *AP* is lower triangular.
trans	CHARACTER*1. Specifies whether or not the copied block of *AP* is transposed. If *trans* = 'N', no transpose: op(*AP*) = *AP*. If *trans* = 'T',transpose: op(*AP*) = AP^T. If *trans* = 'C',conjugate transpose: op(*AP*) = AP^H. For real data this is the same as *trans* = 'T'.
n	INTEGER. The order of the matrix *AP*; $n \geq 0$.
ap	REAL for mkl_stpunpack DOUBLE PRECISION for mkl_dtpunpack COMPLEX for mkl_ctpunpack DOUBLE COMPLEX for mkl_ztpunpack Array, size at least max(1, *n*(*n*+1)/2). The array *ap* contains either the upper or the lower triangular part of the matrix *AP* (as specified by *uplo*) in packed storage (see Matrix Storage Schemes). It is the source for the submatrix of *AP* from row *i* to row *i* + *rows* - 1 and column *j* to column *j* + *cols* - 1 to be copied.
i, j	INTEGER. Coordinates of left upper corner of the submatrix in *AP* to copy. If *uplo*='U', $1 \leq i \leq j \leq n$.

If *uplo*='L', 1 ≤*j*≤*i*≤*n*.

rows INTEGER. Number of rows to copy. 0 ≤*rows*≤*n* - *i* + 1.

cols INTEGER. Number of columns to copy. 0 ≤*cols*≤*n* - *j* + 1.

lda INTEGER. The leading dimension of array *a*.

lda≥ max(1,*rows*) for *trans* = 'N' and and *lda*≥ max(1,*cols*) for *trans*='T' or *trans*='C'.

Output Parameters

a `REAL` for `mkl_stpunpack`

`DOUBLE PRECISION` for `mkl_dtpunpack`

`COMPLEX` for `mkl_ctpunpack`

`DOUBLE COMPLEX` for `mkl_ztpunpack`

Pointer to the destination matrix. The size of *a* must be at least *lda***cols* for *trans* = 'N' or *lda***rows* for *trans*='T' or *trans*='C'. On exit, array *a* is overwritten with a copy of the unpacked *rows*-by-*cols* submatrix of *ap* unpacked rows-by-columns if *trans* = 'N', or unpacked columns-by-rows if *trans* = 'T' or *trans* = 'C'.

NOTE
If there are elements outside of the triangular part of *ap* indicated by *uplo*, they are skipped and are not copied to *a*.

info INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Additional LAPACK Routines

```
call clasyf_aa(uplo, j1, m, nb, a, lda, ipiv, h, ldh, work, info)
call zlasyf_aa(uplo, j1, m, nb, a, lda, ipiv, h, ldh, work, info)
call slasyf_rk(uplo, n, nb, kb, a, lda, e, ipiv, w, ldw, info)
call dlasyf_rk(uplo, n, nb, kb, a, lda, e, ipiv, w, ldw, info)
call clasyf_rk(uplo, n, nb, kb, a, lda, e, ipiv, w, ldw, info)
call zlasyf_rk(uplo, n, nb, kb, a, lda, e, ipiv, w, ldw, info)
call chetf2_rk(uplo, n, a, lda, e, ipiv, info)
call zhetf2_rk(uplo, n, a, lda, e, ipiv, info)
call clahef_rk(uplo, n, nb, kb, a, lda, e, ipiv, w, ldw, info)
call zlahef_rk(uplo, n, nb, kb, a, lda, e, ipiv, w, ldw, info)
call ssytf2_rk(uplo, n, a, lda, e, ipiv, info)
call dsytf2_rk(uplo, n, a, lda, e, ipiv, info)
call csytf2_rk(uplo, n, a, lda, e, ipiv, info)
call zsytf2_rk(uplo, n, a, lda, e, ipiv, info)
```

```
call chetri_3x(uplo, n, a, lda, e, ipiv, work, nb, info)
call zhetri_3x(uplo, n, a, lda, e, ipiv, work, nb, info)
call ssytri_3x(uplo, n, a, lda, e, ipiv, work, nb, info)
call dsytri_3x(uplo, n, a, lda, e, ipiv, work, nb, info)
call csytri_3x(uplo, n, a, lda, e, ipiv, work, nb, info)
call zsytri_3x(uplo, n, a, lda, e, ipiv, work, nb, info)
call ssyconvf(uplo, way, n, a, lda, e, ipiv, info)
call dsyconvf(uplo, way, n, a, lda, e, ipiv, info)
call csyconvf(uplo, way, n, a, lda, e, ipiv, info)
call zsyconvf(uplo, way, n, a, lda, e, ipiv, info)
call ssyconvf_rook(uplo, way, n, a, lda, e, ipiv, info)
call dsyconvf_rook(uplo, way, n, a, lda, e, ipiv, info)
call csyconvf_rook(uplo, way, n, a, lda, e, ipiv, info)
call zsyconvf_rook(uplo, way, n, a, lda, e, ipiv, info)
```

For descriptions of these functions, please see https://www.netlib.org/lapack/explore-html/files.html.

LAPACK Utility Functions and Routines

This section describes LAPACK utility functions and routines.

Summary information about these routines is given in the following table:

LAPACK Utility Routines

Routine Name	Data Types	Description
ilaver		Returns the version of the Lapack library.
ilaenv		Environmental enquiry function which returns values for tuning algorithmic performance.
iparmq		Environmental enquiry function which returns values for tuning algorithmic performance.
ieeeck		Checks if the infinity and NaN arithmetic is safe. Called by ilaenv.
?labad	s, d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
?lamch	s, d	Determines machine parameters for floating-point arithmetic.
?lamc1	s, d	Called from ?lamc2. Determines machine parameters given by *beta*, *t*, *rnd*, *ieee1*.
?lamc2	s, d	Used by ?lamch. Determines machine parameters specified in its arguments list.
?lamc3	s, d	Called from ?lamc1-?lamc5. Intended to force *a* and *b* to be stored prior to doing the addition of *a* and *b*.
?lamc4	s, d	This is a service routine for ?lamc2.
?lamc5	s, d	Called from ?lamc2. Attempts to compute the largest machine floating-point number, without overflow.

Routine Name	Data Types	Description
chla_transtype		Translates a BLAST-specified integer constant to the character string specifying a transposition operation.
iladiag		Translates a character string specifying whether a matrix has a unit diagonal or not to the relevant BLAST-specified integer constant.
ilaprec		Translates a character string specifying an intermediate precision to the relevant BLAST-specified integer constant.
ilatrans		Translates a character string specifying a transposition operation to the BLAST-specified integer constant.
ilauplo		Translates a character string specifying an upper- or lower-triangular matrix to the relevant BLAST-specified integer constant.
xerbla_array		Assists other languages in calling the xerbla function.

See Also

lsame Tests two characters for equality regardless of the case.

lsamen Tests two character strings for equality regardless of the case.

second/dsecnd Returns elapsed time in seconds. Use to estimate real time between two calls to this function.

xerbla Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.

ilaver

Returns the version of the LAPACK library.

Syntax

```
call ilaver( vers_major, vers_minor, vers_patch )
```

Include Files

- mkl.fi

Description

This routine returns the version of the LAPACK library.

Output Parameters

vers_major INTEGER.

Returns the major version of the LAPACK library.

vers_minor INTEGER.

Returns the minor version from the major version of the LAPACK library.

vers_patch INTEGER.

Returns the patch version from the minor version of the LAPACK library.

ilaenv

Environmental enquiry function that returns values for tuning algorithmic performance.

Syntax

```
value = ilaenv( ispec, name, opts, n1, n2, n3, n4 )
```

Include Files

- mkl.fi

Description

The enquiry function `ilaenv` is called from the LAPACK routines to choose problem-dependent parameters for the local environment. See *ispec* below for a description of the parameters.

This version provides a set of parameters that should give good, but not optimal, performance on many of the currently available computers.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

ispec

`INTEGER`.

Specifies the parameter to be returned as the value of `ilaenv`:

= 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance.

= 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used.

= 3: the crossover point (in a block routine, for *n* less than this value, an unblocked routine should be used)

= 4: the number of shifts, used in the nonsymmetric eigenvalue routines (deprecated)

= 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least *k*-by-*m*, where `k` is given by `ilaenv(2,...)` and `m` by `ilaenv(5,...)`

= 6: the crossover point for the SVD (when reducing an *m*-by-*n* matrix to bidiagonal form, if `max(m,n)/min(m,n)` exceeds this value, a *QR* factorization is used first to reduce the matrix to a triangular form.)

= 7: the number of processors

= 8: the crossover point for the multishift *QR* and *QZ* methods for nonsymmetric eigenvalue problems (deprecated).

= 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by `?gelsd` and `?gesdd`)

=10: ieee NaN arithmetic can be trusted not to trap

=11: infinity arithmetic can be trusted not to trap

12 ≤ *ispec* ≤ 16: `?hseqr` or one of its subroutines, see `iparmq` for detailed explanation.

name — `CHARACTER*(*)`. The name of the calling subroutine, in either upper case or lower case.

opts — `CHARACTER*(*)`. The character options to the subroutine *name*, concatenated into a single character string. For example, *uplo* = 'U', *trans* = 'T', and *diag* = 'N' for a triangular routine would be specified as *opts* = 'UTN'.

NOTE
Use only uppercase characters for the *opts* string.

n1, *n2*, *n3*, *n4* — `INTEGER`. Problem dimensions for the subroutine *name*; these may not all be required.

Output Parameters

value — `INTEGER`.

If *value* ≥ 0: the value of the parameter specified by *ispec*;

If *value* = −*k* < 0: the *k*-th argument had an illegal value.

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1*, *n2*, *n3*, *n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by `ilaenv` is checked for validity in the calling subroutine. For example, `ilaenv` is used to retrieve the optimal blocksize for `strtri` as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1> )
if( nb.le.1 ) nb = max( 1, n )
```

See Also

?hseqr

iparmq

iparmq

Environmental enquiry function which returns values for tuning algorithmic performance.

Syntax

value = iparmq(*ispec*, *name*, *opts*, *n*, *ilo*, *ihi*, *lwork*)

Include Files

- mkl.fi

Description

The function sets problem and machine dependent parameters useful for ?hseqr and its subroutines. It is called whenever ilaenv is called with 12≤*ispec*≤16.

Input Parameters

ispec INTEGER.

Specifies the parameter to be returned as the value of iparmq:

= 12: (*inmin*) Matrices of order *nmin* or less are sent directly to ?lahqr, the implicit double shift QR algorithm. *nmin* must be at least 11.

= 13: (*inwin*) Size of the deflation window. This is best set greater than or equal to the number of simultaneous shifts *ns*. Larger matrices benefit from larger deflation windows.

= 14: (*inibl*) Determines when to stop nibbling and invest in an (expensive) multi-shift QR sweep. If the aggressive early deflation subroutine finds *ld* converged eigenvalues from an order *nw* deflation window and *ld*>(*nw***nibble*)/100, then the next QR sweep is skipped and early deflation is applied immediately to the remaining active diagonal block. Setting iparmq(*ispec*=14)=0 causes TTQRE to skip a multi-shift QR sweep whenever early deflation finds a converged eigenvalue. Setting iparmq(*ispec*=14) greater than or equal to 100 prevents TTQRE from skipping a multi-shift QR sweep.

= 15: (*nshfts*) The number of simultaneous shifts in a multi-shift QR iteration.

= 16: (*iacc22*) iparmq is set to 0, 1 or 2 with the following meanings.

0: During the multi-shift QR sweep, ?laqr5 does not accumulate reflections and does not use matrix-matrix multiply to update the far-from-diagonal matrix entries.

1: During the multi-shift QR sweep, ?laqr5 and/or ?laqr3 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.

2: During the multi-shift QR sweep, ?laqr5 accumulates reflections and takes advantage of 2-by-2 block structure during matrix-matrix multiplies.

(If ?trrm is slower than ?gemm, then iparmq(*ispec*=16)=1 may be more efficient than iparmq(*ispec*=16)=2 despite the greater level of arithmetic work implied by the latter choice.)

name CHARACTER*(*). The name of the calling subroutine.

opts CHARACTER*(*). This is a concatenation of the string arguments to TTQRE.

n INTEGER. *n* is the order of the Hessenberg matrix *H*.

ilo, *ihi* INTEGER.

It is assumed that *H* is already upper triangular in rows and columns `1:ilo-1` and `ihi+1:n`.

lwork — INTEGER.

The amount of workspace available.

Output Parameters

value — INTEGER.

If *value* ≥ 0: the value of the parameter specified by *iparmq*;

If *value* = -k < 0: the k-th argument had an illegal value.

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1*, *n2*, *n3*, *n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by `ilaenv` is checked for validity in the calling subroutine. For example, `ilaenv` is used to retrieve the optimal blocksize for `strtri` as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1> )
if( nb.le.1 ) nb = max( 1, n )
```

ieeeck

Checks if the infinity and NaN arithmetic is safe.
Called by `ilaenv`.

Syntax

```
ival = ieeeck( ispec, zero, one )
```

Include Files

- `mkl.fi`

Description

The function `ieeeck` is called from `ilaenv` to verify that infinity and possibly `NaN` arithmetic is safe, that is, will not trap.

Input Parameters

ispec — INTEGER.

Specifies whether to test just for infinity arithmetic or both for infinity and `NaN` arithmetic:

If *ispec* = 0: Verify infinity arithmetic only.

If *ispec* = 1: Verify infinity and NaN arithmetic.

zero — REAL. Must contain the value 0.0

This is passed to prevent the compiler from optimizing away this code.

one — REAL. Must contain the value 1.0

This is passed to prevent the compiler from optimizing away this code.

Output Parameters

ival — INTEGER.

If *ival* = 0: Arithmetic failed to produce the correct answers.

If *ival* = 1: Arithmetic produced the correct answers.

?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call slabad( small, large )
call dlabad( small, large )
```

Include Files

- mkl.fi

Description

The routine takes as input the values computed by slamch/dlamch for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by ?lamch. This subroutine is needed because ?lamch does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

Input Parameters

small — REAL for slabad

DOUBLE PRECISION for dlabad.

The underflow threshold as computed by ?lamch.

large — REAL for slabad

DOUBLE PRECISION for dlabad.

The overflow threshold as computed by ?lamch.

Output Parameters

small — On exit, if log10(*large*) is sufficiently large, the square root of *small*, otherwise unchanged.

large — On exit, if `log10(large)` is sufficiently large, the square root of *large*, otherwise unchanged.

?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
val = slamch( cmach )
val = dlamch( cmach )
```

Include Files

- `mkl.fi`

Description

The function `?lamch` determines single precision and double precision machine parameters.

Input Parameters

cmach — `CHARACTER*1`. Specifies the value to be returned by `?lamch`:

= 'E' or 'e', *val* = *eps*

= 'S' or 's', *val* = *sfmin*

= 'B' or 'b', *val* = *base*

= 'P' or 'p', *val* = *eps***base*

= 'n' or 'n', *val* = *t*

= 'R' or 'r', *val* = *rnd*

= 'M' or 'm', *val* = *emin*

= 'U' or 'u', *val* = *rmin*

= 'L' or 'l', *val* = *emax*

= 'O' or 'o', *val* = *rmax*

where

eps = relative machine precision;

sfmin = safe minimum, such that 1/*sfmin* does not overflow;

base = base of the machine;

prec = *eps***base*;

t = number of (base) digits in the mantissa;

rnd = 1.0 when rounding occurs in addition, 0.0 otherwise;

emin = minimum exponent before (gradual) underflow;

rmin = *underflow_threshold* - *base***(*emin*-1);

emax = largest exponent before overflow;

rmax = *overflow_threshold* - (*base****emax*)*(1-*eps*).

NOTE
You can use a character string for *cmach* instead of a single character in order to make your code more readable. The first character of the string determines the value to be returned. For example, 'Precision' is interpreted as 'p'.

Output Parameters

val
REAL for slamch
DOUBLE PRECISION for dlamch
Value returned by the function.

?lamc1

Called from ?lamc2*. Determines machine parameters given by beta, t, rnd, ieee1.*

Syntax

```
call slamc1( beta, t, rnd, ieee1 )
call dlamc1( beta, t, rnd, ieee1 )
```

Include Files

- mkl.fi

Description

The routine ?lamc1 determines machine parameters given by *beta, t, rnd, ieee1*.

Output Parameters

beta
INTEGER. The base of the machine.

t
INTEGER. The number of (*beta*) digits in the mantissa.

rnd
LOGICAL.
Specifies whether proper rounding (*rnd* = .TRUE.) or chopping (*rnd* = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.

ieee1
LOGICAL.
Specifies whether rounding appears to be done in the *ieee* 'round to nearest' style.

?lamc2

Used by ?lamch*. Determines machine parameters specified in its arguments list.*

Syntax

```
call slamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
call dlamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
```

Include Files

- mkl.fi

Description

The routine ?lamc2 determines machine parameters specified in its arguments list.

Output Parameters

beta	INTEGER. The base of the machine.
t	INTEGER. The number of (*beta*) digits in the mantissa.
rnd	LOGICAL. Specifies whether proper rounding (*rnd* = .TRUE.) or chopping (*rnd* = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.
eps	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest positive number such that *fl*(1.0 - *eps*) < 1.0, where *fl* denotes the computed value.
emin	INTEGER. The minimum exponent before (gradual) underflow occurs.
rmin	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest normalized number for the machine, given by $base^{emin-1}$, where *base* is the floating point value of *beta*.
emax	INTEGER.The maximum exponent before overflow occurs.
rmax	REAL for slamc2 DOUBLE PRECISION for dlamc2 The largest positive number for the machine, given by $base^{emax(1 - eps)}$, where *base* is the floating point value of *beta*.

?lamc3

Called from ?lamc1-?lamc5*. Intended to force a and b to be stored prior to doing the addition of a and b.*

Syntax

val = slamc3(*a*, *b*)

val = dlamc3(*a*, *b*)

Include Files

- mkl.fi

Description

The routine is intended to force *A* and *B* to be stored prior to doing the addition of *A* and *B*, for use in situations where optimizers might hold one of these in a register.

Input Parameters

a, *b*	REAL for slamc3 DOUBLE PRECISION for dlamc3 The values *a* and *b*.

Output Parameters

val	REAL for slamc3 DOUBLE PRECISION for dlamc3 The result of adding values *a* and *b*.

?lamc4

This is a service routine for ?lamc2*.*

Syntax

call slamc4(*emin*, *start*, *base*)

call dlamc4(*emin*, *start*, *base*)

Include Files

- mkl.fi

Description

This is a service routine for ?lamc2.

Input Parameters

start	REAL for slamc4 DOUBLE PRECISION for dlamc4 The starting point for determining *emin*.
base	INTEGER. The base of the machine.

Output Parameters

emin	INTEGER. The minimum exponent before (gradual) underflow, computed by setting *a = start* and dividing by *base* until the previous *a* can not be recovered.

?lamc5

Called from ?lamc2. Attempts to compute the largest machine floating-point number, without overflow.

Syntax

```
call slamc5( beta, p, emin, ieee, emax, rmax)
call dlamc5( beta, p, emin, ieee, emax, rmax)
```

Include Files

- mkl.fi

Description

The routine ?lamc5 attempts to compute *rmax*, the largest machine floating-point number, without overflow. It assumes that *emax + abs(emin)* sum approximately to a power of 2. It will fail on machines where this assumption does not hold, for example, the Cyber 205 (*emin = -28625*, *emax = 28718*). It will also fail if the value supplied for *emin* is too large (that is, too close to zero), probably with overflow.

Input Parameters

beta	INTEGER. The base of floating-point arithmetic.
p	INTEGER. The number of base *beta* digits in the mantissa of a floating-point value.
emin	INTEGER. The minimum exponent before (gradual) underflow.
ieee	LOGICAL. A logical flag specifying whether or not the arithmetic system is thought to comply with the IEEE standard.

Output Parameters

emax	INTEGER. The largest exponent before overflow.
rmax	REAL for slamc5 DOUBLE PRECISION for dlamc5 The largest machine floating-point number.

chla_transtype

Translates a BLAST-specified integer constant to the character string specifying a transposition operation.

Syntax

```
val = chla_transtype( trans )
```

Include Files

- mkl.fi

Description

The chla_transtype function translates a BLAST-specified integer constant to the character string specifying a transposition operation.

The function returns a CHARACTER*1. If the input is not an integer indicating a transposition operator, then *val* is 'X'. Otherwise, the function returns the constant value corresponding to *trans*.

Input Parameters

trans — INTEGER.

Specifies the form of the system of equations:

If *trans* = BLAS_NO_TRANS = 111: No transpose.

If *trans* = BLAS_TRANS = 112: Transpose.

If *trans* = BLAS_CONJ_TRANS = 113: Conjugate Transpose.

Output Parameters

val — CHARACTER*1

Character that specifies a transposition operation.

iladiag

Translates a character string specifying whether a matrix has a unit diagonal to the relevant BLAST-specified integer constant.

Syntax

val = iladiag(*diag*)

Include Files

- mkl.fi

Description

The iladiag function translates a character string specifying whether a matrix has a unit diagonal or not to the relevant BLAST-specified integer constant.

The function returns an INTEGER. If *val* < 0, the input is not a character indicating a unit or non-unit diagonal. Otherwise, the function returns the constant value corresponding to *diag*.

Input Parameters

diag — CHARACTER*1.

Specifies the form of the system of equations:

If *diag* = 'N': *A* is non-unit triangular.

If *diag* = 'U': *A* is unit triangular.

Output Parameters

val	INTEGER Value returned by the function.

ilaprec

Translates a character string specifying an intermediate precision to the relevant BLAST-specified integer constant.

Syntax

val = ilaprec(*prec*)

Include Files

- mkl.fi

Description

The ilaprec function translates a character string specifying an intermediate precision to the relevant BLAST-specified integer constant.

The function returns an INTEGER. If *val* < 0, the input is not a character indicating a supported intermediate precision. Otherwise, the function returns the constant value corresponding to *prec*.

Input Parameters

prec	CHARACTER*1. Specifies the form of the system of equations: If *prec* = 'S': Single. If *prec* = 'D': Double. If *prec* = 'I': Indigenous. If *prec* = 'X', 'E': Extra.

Output Parameters

val	INTEGER Value returned by the function.

ilatrans

Translates a character string specifying a transposition operation to the BLAST-specified integer constant.

Syntax

val = ilatrans(*trans*)

Include Files

- mkl.fi

Description

The `ilatrans` function translates a character string specifying a transposition operation to the BLAST-specified integer constant.

The function returns a `INTEGER`. If *val* < 0, the input is not a character indicating a transposition operator. Otherwise, the function returns the constant value corresponding to *trans*.

Input Parameters

trans — `CHARACTER*1`.

Specifies the form of the system of equations:

If *trans* = `'N'`: No transpose.

If *trans* = `'T'`: Transpose.

If *trans* = `'C'`: Conjugate Transpose.

Output Parameters

val — `INTEGER`

Character that specifies a transposition operation.

ilauplo

Translates a character string specifying an upper- or lower-triangular matrix to the relevant BLAST-specified integer constant.

Syntax

```
val = ilauplo( uplo )
```

Include Files

- `mkl.fi`

Description

The `ilauplo` function translates a character string specifying an upper- or lower-triangular matrix to the relevant BLAST-specified integer constant.

The function returns an `INTEGER`. If *val* < 0, the input is not a character indicating an upper- or lower-triangular matrix. Otherwise, the function returns the constant value corresponding to *uplo*.

Input Parameters

diag — `CHARACTER`.

Specifies the form of the system of equations:

If *diag* = `'U'`: *A* is upper triangular.

If *diag* = `'L'`: *A* is lower triangular.

Output Parameters

val — `INTEGER`

Value returned by the function.

xerbla_array

Assists other languages in calling the `xerbla` *function.*

Syntax

`call xerbla_array( srname_array, srname_len, info )`

Include Files

- `mkl.fi`

Description

The routine assists other languages in calling the error handling `xerbla` function. Rather than taking a Fortran string argument as the function name, `xerbla_array` takes an array of single characters along with the array length. The routine then copies up to 32 characters of that array into a Fortran string and passes that to `xerbla`. If called with a non-positive *`srname_len`*, the routine will call `xerbla` with a string of all blank characters.

If some macro or other device makes `xerbla_array` available to `C99` by a name `lapack_xerbla` and with a common Fortran calling convention, a `C99` program could invoke `xerbla` via:

```
{
   int flen = strlen(__func__);
   lapack_xerbla(__func__, &flen, &info);
}
```

Providing `xerbla_array` is not necessary for intercepting LAPACK errors. `xerbla_array` calls `xerbla`.

Output Parameters

`srname_array` — `CHARACTER(1)`.
Array, dimension (*`srname_len`*). The name of the routine that called `xerbla_array`.

`srname_len` — `INTEGER`.
The length of the name in *`srname_array`*.

`info` — `INTEGER`.
Position of the invalid parameter in the parameter list of the calling routine.

LAPACK Test Functions and Routines

This section describes LAPACK test functions and routines.

?latms

Generates a general m-by-n matrix with specific singular values.

Syntax

`call slatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)`

```
call dlatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
call clatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
call zlatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
```

Description

The ?latms routine generates random matrices with specified singular values, or symmetric/Hermitian matrices with specified eigenvalues for testing LAPACK programs.

It applies this sequence of operations:

1. Set the diagonal to *d*, where *d* is input or computed according to *mode*, *cond*, *dmax*, and *sym* as described in Input Parameters.
2. Generate a matrix with the appropriate band structure, by one of two methods:

Method A	1. Generate a dense *m*-by-*n* matrix by multiplying *d* on the left and the right by random unitary matrices, then: 2. Reduce the bandwidth according to *kl* and *ku*, using Householder transformations.
Method B:	Convert the bandwidth-0 (i.e., diagonal) matrix to a bandwidth-1 matrix using Givens rotations, "chasing" out-of-band elements back, much as in QR; then convert the bandwidth-1 to a bandwidth-2 matrix, etc. Note that for reasonably small bandwidths (relative to *m* and *n*) this requires less storage, as a dense matrix is not generated. Also, for symmetric or Hermitian matrices, only one triangle is generated.

Method A is chosen if the bandwidth is a large fraction of the order of the matrix, and *lda* is at least *m* (so a dense matrix can be stored.) Method B is chosen if the bandwidth is small (less than (1/2)*n for symmetric or Hermitian or less than .3*n+m for nonsymmetric), or *lda* is less than *m* and not less than the bandwidth.

Pack the matrix if desired, using one of the methods specified by the *pack* parameter.

If Method B is chosen and band format is specified, then the matrix is generated in the band format and no repacking is necessary.

Input Parameters

The data types are given for the Fortran interface.

m	INTEGER. The number of rows of the matrix *A* (*m*≥ 0).
n	INTEGER. The number of columns of the matrix *A* (*n*≥ 0).
dist	CHARACTER*1. Specifies the type of distribution to be used to generate the random singular values or eigenvalues: • 'U': uniform distribution (0, 1) • 'S': symmetric uniform distribution (-1, 1) • 'N': normal distribution (0, 1)

iseed

INTEGER. Array with size 4.

Specifies the seed of the random number generator. Values should lie between 0 and 4095 inclusive, and *iseed*(4) should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers. The values of the array are modified, and can be used in the next call to ?latms to continue the same random number sequence.

sym

CHARACTER*1.

If *sym*='S' or 'H', the generated matrix is symmetric or Hermitian, with eigenvalues specified by *d*, *cond*, *mode*, and *dmax*; they can be positive, negative, or zero.

If *sym*='P', the generated matrix is symmetric or Hermitian, with eigenvalues (which are singular, non-negative values) specified by *d*, *cond*, *mode*, and *dmax*.

If *sym*='N', the generated matrix is nonsymmetric, with singular, non-negative values specified by *d*, *cond*, *mode*, and *dmax*.

d

REAL for slatms and clatms

DOUBLE PRECISION for dlatms and zlatms

Array, size (MIN(*m* , *n*))

This array is used to specify the singular values or eigenvalues of *A* (see the description of *sym*). If *mode*=0, then *d* is assumed to contain the eigenvalues or singular values, otherwise elements of *d* are computed according to *mode*, *cond*, and *dmax*.

mode

INTEGER. Describes how the singular/eigenvalues are specified.

- *mode* = 0: use *d* as input
- *mode* = 1: set *d*(1) = 1 and *d*(2:*n*) = 1.0/*cond*
- *mode* = 2: set *d*(1:*n* - 1) = 1 and *d*(*n*) = 1.0/*cond*
- *mode* = 3: set *d*(*i*) = $cond^{-(i - 1)/(n - 1)}$
- *mode* = 4: set *d*(*i*) = 1 - (*i* - 1)/(*n* - 1)*(1 - 1/*cond*)
- *mode* = 5: set elements of *d* to random numbers in the range (1/*cond* , 1) such that their logarithms are uniformly distributed.
- *mode* = 6: set elements of *d* to random numbers from same distribution as the rest of the matrix.

mode < 0 has the same meaning as ABS(*mode*), except that the order of the elements of *d* is reversed. Thus, if *mode* is positive, *d* has entries ranging from 1 to 1/*cond*, if negative, from 1/*cond* to 1.

If *sym*='S' or 'H', and *mode* is not 0, 6, nor -6, then the elements of *d* are also given a random sign (multiplied by +1 or -1).

cond

REAL for slatms and clatms

DOUBLE PRECISION for dlatms and zlatms

Used in setting *d* as described for the *mode* parameter. If used, *cond*≥ 1.

dmax

REAL for slatms and clatms

DOUBLE PRECISION for dlatms and zlatms

If *mode* is not -6, 0 nor 6, the contents of *d*, as computed according to *mode* and *cond*, are scaled by *dmax* / max(abs(*d*(*i*))); thus, the maximum absolute eigenvalue or singular value (the norm) is abs(*dmax*).

NOTE
dmax need not be positive: if *dmax* is negative (or zero), *d* will be scaled by a negative number (or zero).

kl INTEGER. Specifies the lower bandwidth of the matrix. For example, *kl*=0 implies upper triangular, *kl*=1 implies upper Hessenberg, and *kl* being at least *m* - 1 means that the matrix has full lower bandwidth. *kl* must equal *ku* if the matrix is symmetric or Hermitian.

ku INTEGER. Specifies the upper bandwidth of the matrix. For example, *ku*=0 implies lower triangular, *ku*=1 implies lower Hessenberg, and *ku* being at least *n* - 1 means that the matrix has full upper bandwidth. *kl* must equal *ku* if the matrix is symmetric or Hermitian.

pack CHARACTER*1. Specifies packing of matrix:

- 'N': no packing
- 'U': zero out all subdiagonal entries (if symmetric or Hermitian)
- 'L': zero out all superdiagonal entries (if symmetric or Hermitian)
- 'B': store the lower triangle in band storage scheme (only if matrix symmetric, Hermitian, or lower triangular)
- 'Q': store the upper triangle in band storage scheme (only if matrix symmetric, Hermitian, or upper triangular)
- 'Z': store the entire matrix in band storage scheme (pivoting can be provided for by using this option to store *A* in the trailing rows of the allocated storage)

Using these options, the various LAPACK packed and banded storage schemes can be obtained:

	'Z'	'B'	'Q'	'C'	'R'
GB: general band	x				
PB: symmetric positive definite band		x	x		
SB: symmetric band		x	x		
HB: Hermitian band		x	x		
TB: triangular band		x	x		
PP: symmetric positive definite packed				x	x
SP: symmetric packed				x	x
HP: Hermitian packed				x	x
TP: triangular packed				x	x

If two calls to ?latms differ only in the *pack* parameter, they generate mathematically equivalent matrices.

lda INTEGER. *lda* specifies the first dimension of *a* as declared in the calling program.

If *pack*='N', 'U', 'L', 'C', or 'R', then *lda* must be at least *m*.

If *pack*='B' or 'Q', then *lda* must be at least MIN(*kl*, *m* – 1) (which is equal to MIN(*ku*,*n* - 1)).

If *pack*='Z', *lda* must be large enough to hold the packed array: MIN(*ku*, *n* – 1) + MIN(*kl*, *m* – 1) + 1.

Output Parameters

iseed The array *iseed* contains the updated seed.

d The array *d* contains the updated seed.

> **NOTE**
> The array *d* is not modified if *mode* = 0.

a REAL for slatms

DOUBLE PRECISION for dlatms

COMPLEX for clatms

DOUBLE COMPLEX for zlatms

Array of size *lda* by *n*.

The array *a* contains the generated *m*-by-*n* matrix *A*.

a is first generated in full (unpacked) form, and then packed, if so specified by *pack*. Thus, the first *m* elements of the first *n* columns are always modified. If *pack* specifies a packed or banded storage scheme, all *lda* elements of the first *n* columns are modified; the elements of the array which do not correspond to elements of the generated matrix are set to zero.

work REAL for slatms

DOUBLE PRECISION for dlatms

COMPLEX for clatms

DOUBLE COMPLEX for zlatms

Array of size (3*MAX(*n*, *m*))

Workspace.

info INTEGER. If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

If *info* = 2, cannot scale to *dmax* (maximum singular value is 0).

If *info* = 3, error return from ?lagge, ?laghe, or ?lagsy.

Additional LAPACK Routines (Included for Compatibility with Netlib LAPACK)

```
call chesv_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , b , ldb ,
info);
```

```
call dsysv_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , b , ldb ,
info);
```

```
call ssysv_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , b , ldb ,
info);
```

```
call zhesv_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , * ipiv2 , b , ldb ,
info);
```

```
call chetrf_aa_2stage (uplo , n , a , lda , tb , ltb , ipiv , ipiv2 , info);
```

```
call dsytrf_aa_2stage (uplo , n , a , lda , tb , ltb , ipiv , ipiv2 , info);
```

```
call ssytrf_aa_2stage (uplo , n , a , lda , tb , ltb , ipiv , ipiv2 , info);
```

```
call zhetrf_aa_2stage (uplo , n , a , lda , tb , ltb , ipiv , ipiv2 , info);
```

```
call chetrs_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , b , ldb ,
info);
```

```
call dsytrs_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , b , ldb ,
info);
```

```
call ssytrs_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , b , ldb ,
info);
```

```
call zhetrs_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , b , ldb ,
info);
```

```
call csytrf_aa_2stage (uplo , n , a , lda , tb , ltb , ipiv , ipiv2 , info);
```

```
call zsytrf_aa_2stage (uplo , n , a , lda , tb , ltb , ipiv , ipiv2 , info);
```

```
call csytrs_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , * b , ldb ,
info);
```

```
call zsytrs_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , * b , ldb ,
info);
```

```
call ssyevd_2stage(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
```

```
call dsyevd_2stage(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
```

```
call ssyevr_2stage(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, iwork, liwork, info)
```

```
call dsyevr_2stage(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, iwork, liwork, info)
```

```
call ssyevx_2stage(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
work, lwork, iwork, ifail, info)
```

```
call dsyevx_2stage(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
work, lwork, iwork, ifail, info)
```

```
call ssygv_2stage(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
```

```
call dsygv_2stage(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
```

```
call cheev_2stage (jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
```

```
call zheev_2stage(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
```

```
call cheevd_2stage(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork,
info)
```

call zheevd_2stage(*jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call cheevr_2stage(*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call zheevr_2stage(*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call cheevx_2stage(*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *iwork*, *ifail*, *info*)

call zheevx_2stage(*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *iwork*, *ifail*, *info*)

call chegv_2stage(*itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*, *work*, *lwork*, *rwork*, *info*)

call zhegv_2stage(*itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*, *work*, *lwork*, *rwork*, *info*)

call ssbev_2stage(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *info*)

call dsbev_2stage(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *info*)

call ssbevd_2stage(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call dsbevd_2stage(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call ssbevx_2stage(*jobz*, *range*, *uplo*, *n*, *kd*, *ab*, *ldab*, *q*, *ldq*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *iwork*, *ifail*, *info*)

call dsbevx_2stage(*jobz*, *range*, *uplo*, *n*, *kd*, *ab*, *ldab*, *q*, *ldq*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *iwork*, *ifail*, *info*)

call ssytrd_2stage(*vect*, *uplo*, *n*, *a*, *lda*, *d*, *e*, *tau*, *hous2*, *lhous2*, *work*, *lwork*, *info*)

call dsytrd_2stage(*vect*, *uplo*, *n*, *a*, *lda*, *d*, *e*, *tau*, *hous2*, *lhous2*, *work*, *lwork*, *info*)

call chbev_2stage(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *info*)

call zhbev_2stage(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *info*)

call chbevd_2stage(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call zhbevd_2stage(*jobz*, *uplo*, *n*, *kd*, *ab*, *ldab*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call chbevx_2stage(*jobz*, *range*, *uplo*, *n*, *kd*, *ab*, *ldab*, *q*, *ldq*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *iwork*, *ifail*, *info*)

call zhbevx_2stage(*jobz*, *range*, *uplo*, *n*, *kd*, *ab*, *ldab*, *q*, *ldq*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *iwork*, *ifail*, *info*)

call chetrd_2stage(*vect*, *uplo*, *n*, *a*, *lda*, *d*, *e*, *tau*, *hous2*, *lhous2*, *work*, *lwork*, *info*)

call zhetrd_2stage(*vect*, *uplo*, *n*, *a*, *lda*, *d*, *e*, *tau*, *hous2*, *lhous2*, *work*, *lwork*, *info*)

call chetrd_hb2st(*stage1*, *vect*, *uplo*, *n*, *kd*, *ab*, *ldab*, *d*, *e*, *hous*, *lhous*, *work*, *lwork*, *info*)

call zhetrd_hb2st(*stage1*, *vect*, *uplo*, *n*, *kd*, *ab*, *ldab*, *d*, *e*, *hous*, *lhous*, *work*, *lwork*, *info*)

call chetrd_he2hb(*uplo*, *n*, *kd*, *a*, *lda*, *ab*, *ldab*, *tau*, *work*, *lwork*, *info*)

call zhetrd_he2hb(*uplo*, *n*, *kd*, *a*, *lda*, *ab*, *ldab*, *tau*, *work*, *lwork*, *info*)

call chb2st_kernels(*uplo*, *wantz*, *ttype*, *st*, *ed*, *sweep*, *n*, *nb*, *ib*, *a*, *lda*, *v*, *tau*, *ldvt*, *work*)

```
call zhb2st_kernels(uplo, wantz, ttype, st, ed, sweep, n, nb, ib, a, lda, v, tau, ldvt,
work)
```

```
call ssb2st_kernels(uplo, wantz, ttype, st, ed, sweep, n, nb, ib, a, lda, v, tau, ldvt,
work)
```

```
call dsb2st_kernels(uplo, wantz, ttype, st, ed, sweep, n, nb, ib, a, lda, v, tau, ldvt,
work)
```

```
call iparam2stage(ispec, name, opts, ni, nbi, ibi, nxi)
```

For descriptions of these functions, please see https://www.netlib.org/lapack/explore-html/files.html.

ScaLAPACK Routines

Intel® oneAPI Math Kernel Library implements routines from the ScaLAPACK package for distributed-memory architectures. Routines are supported for both real and complex dense and band matrices to perform the tasks of solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

Intel® oneAPI Math Kernel Library (oneMKL) ScaLAPACK routines are written in FORTRAN 77 with exception of a few utility routines written in C to exploit the IEEE arithmetic. All routines are available in all precision types: single precision, double precision, complexm, and double complex precision. See the`mkl_scalapack.h` header file for C declarations of ScaLAPACK routines.

> **NOTE**
> ScaLAPACK routines are provided only for Intel® 64 or Intel® Many Integrated Core architectures.

See descriptions of ScaLAPACK computational routines that perform distinct computational tasks, as well as driver routinesfor solving standard types of problems in one call. Additionally, Intel® oneAPI Math Kernel Library implements ScaLAPACKAuxiliary Routines, Utility Functions and Routines, and Matrix Redistribution/Copy Routines. The library includes routines for both real and complex data.

The `<install_directory>/examples/scalapackf` directory contains sample code demonstrating the use of ScaLAPACK routines.

Generally, ScaLAPACK runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of BLAS optimized for the target architecture. Intel® oneAPI Math Kernel Library (oneMKL) version of ScaLAPACK is optimized for Intel® processors. For the detailed system and environment requirements, see*Intel® oneAPI Math Kernel Library (oneMKL) Release Notes* and *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

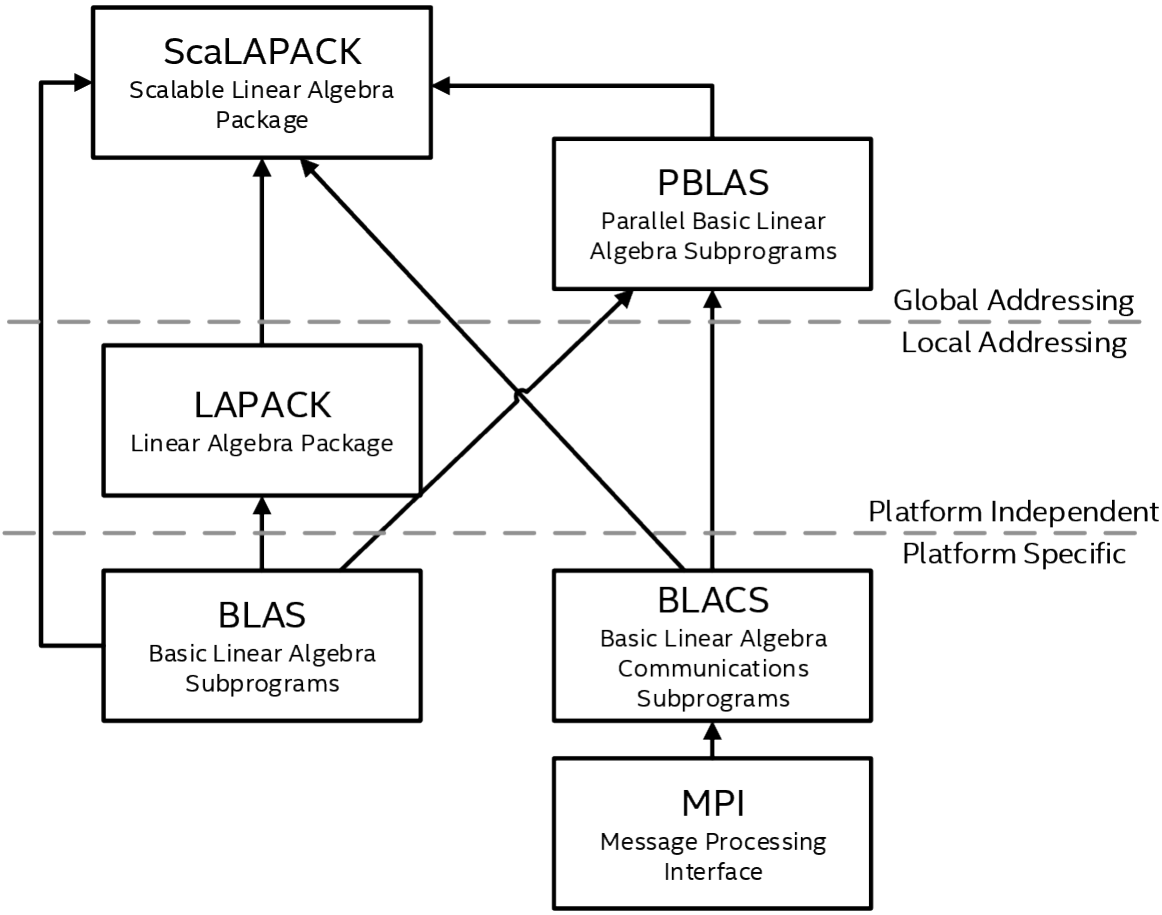
For full reference on ScaLAPACK routines and related information, see [SLUG].

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Overview of ScaLAPACK Routines

The model of the computing environment for ScaLAPACK is represented as a one-dimensional array of processes (for operations on band or tridiagonal matrices) or also a two-dimensional process grid (for operations on dense matrices). To use ScaLAPACK, all global matrices or vectors should be distributed on this array or grid prior to calling the ScaLAPACK routines.

ScaLAPACK is closely tied to other components, including BLAS, BLACS, LAPACK, and PBLAS.



ScaLAPACK Array Descriptors

ScaLAPACK uses two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, and also allows use of BLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global matrix (array) and its corresponding process and memory location is contained in the array called the *array descriptor* associated with each global matrix. The size of the array descriptor is denoted as *dlen_*.

Let *A* be a two-dimensional block cyclicly distributed matrix with the array descriptor array `desca`. The meaning of each array descriptor element depends on the type of the matrix *A*. The tables "Array descriptor for dense matrices" and "Array descriptor for narrow-band and tridiagonal matrices" describe the meaning of each element for the different types of matrices.

Array descriptor for dense matrices (*dlen_*=9)

Element Name	Stored in	Description	Element Index Number
dtype_a	`desca`(*dtype_*)	Descriptor type (=1 for dense matrices).	1
ctxt_a	`desca`(*ctxt_*)	BLACS context handle for the process grid.	2
m_a	`desca`(*m_*)	Number of rows in the global matrix *A*.	3
n_a	`desca`(*n_*)	Number of columns in the global matrix *A*.	4
mb_a	`desca`(*mb_*)	Row blocking factor.	5

Element Name	Stored in	Description	Element Index Number
nb_a	*desca*(*nb_*)	Column blocking factor.	6
rsrc_a	*desca*(*rsrc_*)	Process row over which the first row of the global matrix *A* is distributed.	7
csrc_a	*desca*(*csrc_*)	Process column over which the first column of the global matrix *A* is distributed.	8
lld_a	*desca*(*lld_*)	Leading dimension of the local matrix *A*.	9

Array descriptor for narrow-band and tridiagonal matrices (*dlen_*=7)

Element Name	Stored in	Description	Element Index Number
dtype_a	*desca*(*dtype_*)	Descriptor type • *dtype_a*=501: 1-by-*P* grid, • *dtype_a*=502: *P*-by-1 grid.	1
ctxt_a	*desca*(*ctxt_*)	BLACS context handle indicating the BLACS process grid over which the global matrix *A* is distributed. The context itself is global, but the handle (the integer value) can vary.	2
n_a	*desca*(*n_*)	The size of the matrix dimension being distributed.	3
nb_a	*desca*(*nb_*)	The blocking factor used to distribute the distributed dimension of the matrix *A*.	4
src_a	*desca*(*src_*)	The process row or column over which the first row or column of the matrix *A* is distributed.	5
lld_a	*desca*(*lld_*)	The leading dimension of the local matrix storing the local blocks of the distributed matrix *A*. The minimum value of *lld_a* depends on *dtype_a*. • *dtype_a*=501: *lld_a*≥ max(size of undistributed dimension, 1), • *dtype_a*=502: *lld_a*≥ max(*nb_a*, 1).	6
Not applicable		Reserved for future use.	7

Similar notations are used for different matrices. For example: *lld_b* is the leading dimension of the local matrix storing the local blocks of the distributed matrix *B* and *dtype_z* is the type of the global matrix *Z*.

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by $LOC_r()$ and $LOC_c()$, respectively. To compute these numbers, you can use the ScaLAPACK tool routine `numroc`.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix sub(*A*) of the global matrix *A* defined by the following 6 values (for dense matrices):

m	The number of rows of sub(*A*)
n	The number of columns of sub(*A*)
a	A pointer to the local matrix containing the entire global matrix A
ia	The row index of sub(*A*) in the global matrix *A*
ja	The column index of sub(*A*) in the global matrix *A*

desca	The array descriptor for the global matrix *A*

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Naming Conventions for ScaLAPACK Routines

For each routine introduced in this chapter, you can use the ScaLAPACK name. The naming convention for ScaLAPACK routines is similar to that used for LAPACK routines. A general rule is that each routine name in ScaLAPACK, which has an LAPACK equivalent, is simply the LAPACK name prefixed by initial letter `p`.

ScaLAPACK names have the structure `p?yyzzz` or `p?yyzz`, which is described below.

The initial letter `p` is a distinctive prefix of ScaLAPACK routines and is present in each such routine.

The second symbol `?` indicates the data type:

`s`	real, single precision
`d`	real, double precision
`c`	complex, single precision
`z`	complex, double precision

The second and third letters `yy` indicate the matrix type as:

`ge`	general
`gb`	general band
`gg`	a pair of general matrices (for a generalized problem)
`dt`	general tridiagonal (diagonally dominant-like)
`db`	general band (diagonally dominant-like)
`po`	symmetric or Hermitian positive-definite
`pb`	symmetric or Hermitian positive-definite band
`pt`	symmetric or Hermitian positive-definite tridiagonal
`sy`	symmetric
`st`	symmetric tridiagonal (real)
`he`	Hermitian
`or`	orthogonal
`tr`	triangular (or quasi-triangular)
`tz`	trapezoidal
`un`	unitary

For computational routines, the last three letters **zzz** indicate the computation performed and have the same meaning as for LAPACK routines.

For driver routines, the last two letters **zz** or three letters **zzz** have the following meaning:

sv	a *simple* driver for solving a linear system
svx	an *expert* driver for solving a linear system
ls	a driver for solving a linear least squares problem
ev	a simple driver for solving a symmetric eigenvalue problem
evd	a simple driver for solving an eigenvalue problem using a divide and conquer algorithm
evx	an expert driver for solving a symmetric eigenvalue problem
svd	a driver for computing a singular value decomposition
gvx	an expert driver for solving a generalized symmetric definite eigenvalue problem

Simple driver here means that the driver just solves the general problem, whereas an *expert* driver is more versatile and can also optionally perform some related computations (such, for example, as refining the solution and computing error bounds after the linear system is solved).

ScaLAPACK Computational Routines

In the sections that follow, the descriptions of ScaLAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

- Solving Systems of Linear Equations
- Orthogonal Factorizations and LLS Problems
- Symmetric Eigenproblems
- Nonsymmetric Eigenproblems
- Singular Value Decomposition
- Generalized Symmetric-Definite Eigenproblems

See also the respective driver routines.

Systems of Linear Equations: ScaLAPACK Computational Routines

ScaLAPACK supports routines for the systems of equations with the following types of matrices:

- general
- general banded
- general diagonally dominant-like banded (including general tridiagonal)
- symmetric or Hermitian positive-definite
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian positive-definite tridiagonal

A *diagonally dominant-like* matrix is defined as a matrix for which it is known in advance that pivoting is not required in the *LU* factorization of this matrix.

For the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix; *equilibrating* the matrix; *solving* a system of linear equations; *estimating the condition number* of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix. Note that for some of the listed matrix types only part of the computational routines are provided (for example, routines that refine the solution are not provided for band or tridiagonal matrices). See Table "Computational Routines for Systems of Linear Equations" for full list of available routines.

To solve a particular problem, you can either call two or more computational routines or call a corresponding driver routine that combines several tasks in one call. Thus, to solve a system of linear equations with a general matrix, you can first call `p?getrf`(*LU* factorization) and then `p?getrs`(computing the solution). Then, you might wish to call `p?gerfs` to refine the solution and get the error bounds. Alternatively, you can just use the driver routine `p?gesvx` which performs all these tasks in one call.

Table "Computational Routines for Systems of Linear Equations" lists the ScaLAPACK computational routines for factorizing, equilibrating, and inverting matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.

Computational Routines for Systems of Linear Equations

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general (partial pivoting)	p?getrf	p?geequ	p?getrs	p?gecon	p?gerfs	p?getri
general band (partial pivoting)	p?gbtrf		p?gbtrs			
general band (no pivoting)	p?dbtrf		p?dbtrs			
general tridiagonal (no pivoting)	p?dttrf		p?dttrs			
symmetric/Hermitian positive-definite	p?potrf	p?poequ	p?potrs	p?pocon	p?porfs	p?potri
symmetric/Hermitian positive-definite, band	p?pbtrf		p?pbtrs			
symmetric/Hermitian positive-definite, tridiagonal	p?pttrf		p?pttrs			
triangular			p?trtrs	p?trcon	p?trrfs	p?trtri

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex).

Matrix Factorization: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization of general matrices
- LU factorization of diagonally dominant-like matrices
- Cholesky factorization of real symmetric or complex Hermitian positive-definite matrices

You can compute the factorizations using full and band storage of matrices.

p?getrf

Computes the LU factorization of a general m-by-n distributed matrix.

Syntax

```
call psgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pdgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pcgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pzgetrf(m, n, a, ia, ja, desca, ipiv, info)
```

Include Files

Description

The p?getrfroutine forms the *LU* factorization of a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) as

A = *P***L***U*

where *P* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if *m*>*n*) and *U* is upper triangular (upper trapezoidal if *m* < *n*). *L* and *U* are stored in sub(*A*).

The routine uses partial pivoting, with row interchanges.

NOTE
This routine supports the Progress Routine feature. See mkl_progress for details.

Input Parameters

m (global) INTEGER. The number of rows in the distributed matrix sub(*A*); *m*≥0.

n (global) INTEGER. The number of columns in the distributed matrix sub(*A*); *n*≥0.

a (local)

REAL for psgetrf

DOUBLE PRECISION for pdgetrf

COMPLEX for pcgetrf

DOUBLE COMPLEX for pzgetrf.

Pointer into the local memory to an array of local size (*lld_a*, *LOCc*(*ja*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a Overwritten by local pieces of the factors *L* and *U* from the factorization *A* = *P***L***U*. The unit diagonal elements of *L* are not stored.

ipiv (local) INTEGER Array of size *LOCr*(*m_a*)+ *mb_a*.

Contains the pivoting information: local row *i* was interchanged with global row *ipiv*(*i*). This array is tied to the distributed matrix *A*.

info (global) INTEGER.

If *info*=0, the execution is successful.

info < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* = *i* > 0, $u_{ia+i,\ ja+j-1}$ is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gbtrf

Computes the LU factorization of a general n-by-n banded distributed matrix.

Syntax

```
call psgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pdgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pcgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pzgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
```

Include Files

Description

The `p?gbtrf` routine computes the *LU* factorization of a general *n*-by-*n* real/complex banded distributed matrix *A*(1:*n*, *ja*:*ja*+*n*-1) using partial pivoting with row interchanges.

The resulting factorization is not the same factorization as returned from the LAPACK routine `?gbtrf`. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form

A(1:*n*, *ja*:*ja*+*n*-1) = *P***L***U***Q*

where *P* and *Q* are permutation matrices, and *L* and *U* are banded lower and upper triangular matrices, respectively. The matrix *Q* represents reordering of columns for the sake of parallelism, while *P* represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

n — (global) `INTEGER`. The number of rows and columns in the distributed submatrix *A*(1:*n*, *ja*:*ja*+*n*-1); *n*≥ 0.

bwl — (global) `INTEGER`. The number of sub-diagonals within the band of *A* (0 ≤ *bwl* ≤ *n-1*).

bwu — (global) `INTEGER`. The number of super-diagonals within the band of *A* (0 ≤ *bwu* ≤ *n-1*).

a — (local)

`REAL` for `psgbtrf`

`DOUBLE PRECISION` for `pdgbtrf`

`COMPLEX` for `pcgbtrf`

`DOUBLE COMPLEX` for `pzgbtrf`.

Pointer into the local memory to an array of local size `(lld_a, LOCc(ja+n-1))` where

lld_a≥ 2*bwl + 2*bwu +1.

Contains the local pieces of the *n*-by-*n* distributed banded matrix *A*(1:*n*, *ja*:*ja*+*n*-1) to be factored.

ja

(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501, then *dlen_*≥ 7;

else if *dtype_a* = 1, then *dlen_*≥ 9.

laf

(local) INTEGER. The size of the array *af*.

Must be *laf*≥ (nb_a+*bwu*)*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2*bwu).

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work

(local) Same type as *a*. Workspace array of size *lwork*.

lwork

(local or global) INTEGER. The size of the *work* array (*lwork*≥ 1). If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

a

On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

ipiv

(local) INTEGER array.

The size of *ipiv* must be ≥*nb_a*.

Contains pivot indices for local factorizations. Note that you *should not alter* the contents of this array between factorization and solve.

af

(local)

REAL for psgbtrf

DOUBLE PRECISION for pdgbtrf

COMPLEX for pcgbtrf

DOUBLE COMPLEX for pzgbtrf.

Array of size *laf*.

Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?gbtrf and is stored in *af*.

Note that if a linear system is to be solved using p?gbtrs after the factorization routine,*af* must not be altered after the factorization.

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required.

info

(global) INTEGER.

If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info> 0:

If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not nonsingular, and the factorization was not completed.

If *info* = *k* > NPROCS, the submatrix stored on processor *info*–NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrf

Computes the LU factorization of a n-by-n diagonally dominant-like banded distributed matrix.

Syntax

```
call psdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pddbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pcdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pzdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
```

Include Files

Description

The p?dbtrfroutine computes the LU factorization of a *n*-by-*n* real/complex diagonally dominant-like banded distributed matrix *A*(1:*n*, *ja*:*ja*+*n*-1) without pivoting.

NOTE
A matrix is called *diagonally dominant-like* if pivoting is not required for LU to be numerically stable.

Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

Input Parameters

n
(global) INTEGER. The number of rows and columns in the distributed submatrix *A*(1:*n*, *ja*:*ja*+*n*-1); *n*≥ 0.

bwl
(global) INTEGER. The number of sub-diagonals within the band of *A* (0 ≤ *bwl* ≤ *n-1*).

bwu
(global) INTEGER. The number of super-diagonals within the band of *A* (0 ≤ *bwu* ≤ *n-1*).

a
(local)

REAL for psdbtrf

DOUBLE PRECISION for pddbtrf

COMPLEX for pcdbtrf

DOUBLE COMPLEX for pzdbtrf.

Pointer into the local memory to an array of local size (*lld_a*, *LOCc*(*ja*+*n*-1)).

Contains the local pieces of the *n*-by-*n* distributed banded matrix *A*(1:*n*, *ja*:*ja*+*n*-1) to be factored.

ja
(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca
(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501, then *dlen_*≥ 7;

else if *dtype_a* = 1, then *dlen_*≥ 9.

laf
(local) INTEGER. The size of the array *af*.

Must be *laf*≥NB*(*bwl*+*bwu*)+6*(max(*bwl*,*bwu*))^2 .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work
(local) Same type as *a*. Workspace array of size *lwork*.

lwork
(local or global) INTEGER. The size of the *work* array, must be *lwork*≥ (max(*bwl*,*bwu*))^2. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

a
On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

af
(local)

REAL for psdbtrf

DOUBLE PRECISION for pddbtrf

COMPLEX for pcdbtrf

DOUBLE COMPLEX for pzdbtrf.

Array of size *laf*.

Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?dbtrf and is stored in *af*.

Note that if a linear system is to be solved using p?dbtrs after the factorization routine,*af* must not be altered after the factorization.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info> 0:

If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not diagonally dominant-like, and the factorization was not completed.

If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dttrf

Computes the LU factorization of a diagonally dominant-like tridiagonal distributed matrix.

Syntax

```
call psdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pddttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pcdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pzdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
```

Include Files

Description

The p?dttrfroutine computes the *LU* factorization of an *n*-by-*n* real/complex diagonally dominant-like tridiagonal distributed matrix *A*(1:*n*, *ja*:*ja*+*n*-1) without pivoting for stability.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$A(1:n, ja:ja+n-1) = P*L*U*P^T$,

where P is a permutation matrix, and L and U are banded lower and upper triangular matrices, respectively.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

n

(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$ ($n \geq 0$).

dl, *d*, *du*

(local)

REAL for pspttrf

DOUBLE PRECISON for pdpttrf

COMPLEX for pcpttrf

DOUBLE COMPLEX for pzpttrf.

Pointers to the local arrays of size *nb_a* each.

On entry, the array *dl* contains the local part of the global vector storing the subdiagonal elements of the matrix. Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*.

On entry, the array *d* contains the local part of the global vector storing the diagonal elements of the matrix.

On entry, the array *du* contains the local part of the global vector storing the super-diagonal elements of the matrix. *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja

(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501, then *dlen_* ≥ 7;

else if *dtype_a* = 1, then *dlen_* ≥ 9.

laf

(local) INTEGER. The size of the array *af*.

Must be *laf* ≥ 2*(NB+2) .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work

(local) Same type as *d*. Workspace array of size *lwork*.

lwork

(local or global) INTEGER. The size of the *work* array, must be at least *lwork* ≥ 8*NPCOL.

Output Parameters

dl, *d*, *du*	On exit, overwritten by the information containing the factors of the matrix.
af	(local) REAL for psdttrf DOUBLE PRECISION for pddttrf COMPLEX for pcdttrf DOUBLE COMPLEX for pzdttrf. Array of size *laf*. Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?dttrf and is stored in *af*. Note that if a linear system is to be solved using p?dttrs after the factorization routine,*af* must not be altered.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. *info*> 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not diagonally dominant-like, and the factorization was not completed. If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite distributed matrix.

Syntax

```
call pspotrf(uplo, n, a, ia, ja, desca, info)
call pdpotrf(uplo, n, a, ia, ja, desca, info)
call pcpotrf(uplo, n, a, ia, ja, desca, info)
call pzpotrf(uplo, n, a, ia, ja, desca, info)
```

Include Files

Description

The p?potrfroutine computes the Cholesky factorization of a real symmetric or complex Hermitian positive-definite distributed *n*-by-*n* matrix *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), denoted below as sub(*A*).

The factorization has the form

sub(*A*) = U^H*U if *uplo*='U', or

sub(*A*) = $L*L^H$ if *uplo*='L'

where *L* is a lower triangular matrix and *U* is upper triangular.

Input Parameters

uplo

(global) CHARACTER*1.

Indicates whether the upper or lower triangular part of sub(*A*) is stored. Must be 'U' or 'L'.

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix sub(*A*) that is factored as U^H*U.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix sub(*A*) that is factored as $L*L^H$.

n

(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).

a

(local)

REAL for pspotrf

DOUBLE PRECISON for pdpotrf

COMPLEX for pcpotrf

DOUBLE COMPLEX for pzpotrf.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the *n*-by-*n* symmetric/ Hermitian distributed matrix sub(*A*) to be factored.

Depending on *uplo*, the array *a* contains either the upper or the lower triangular part of the matrix sub(*A*) (see *uplo*).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a

The upper or lower triangular part of *a* is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.

info

(global) INTEGER.

If *info*=0, the execution is successful;

info < 0: if the *i*-th argument is an array, and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* = *k* >0, the leading minor of order *k*, *A*(*ia*:*ia*+*k*-1, *ja*:*ja*+*k*-1), is not positive-definite, and the factorization could not be completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite banded distributed matrix.

Syntax

```
call pspbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pdpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pcpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pzpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
```

Include Files

Description

The p?pbtrfroutine computes the Cholesky factorization of an *n*-by-*n* real symmetric or complex Hermitian positive-definite banded distributed matrix *A*(1:*n*, *ja*:*ja*+*n*-1).

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

A(1:*n*, *ja*:*ja*+*n*-1) = $P*U^H*U*P^T$, if *uplo*='U', or

A(1:*n*, *ja*:*ja*+*n*-1) = $P*L*L^H*P^T$, if *uplo*='L',

where *P* is a permutation matrix and *U* and *L* are banded upper and lower triangular matrices, respectively.

Input Parameters

uplo

(global) CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', upper triangle of *A*(1:*n*, *ja*:*ja*+*n*-1) is stored;

If *uplo* = 'L', lower triangle of *A*(1:*n*, *ja*:*ja*+*n*-1) is stored.

n

(global) INTEGER. The order of the distributed submatrix *A*(1:*n*, *ja*:*ja*+*n*-1).

(*n*≥0).

bw (global) INTEGER.

The number of superdiagonals of the distributed matrix if *uplo* = 'U', or the number of subdiagonals if *uplo* = 'L' (*bw*≥0).

a (local)

REAL for pspbtrf

DOUBLE PRECISON for pdpbtrf

COMPLEX for pcpbtrf

DOUBLE COMPLEX for pzpbtrf.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the upper or lower triangle of the symmetric/Hermitian band distributed matrix *A*(1:*n*, *ja*:*ja*+*n*-1) to be factored.

ja (global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501, then *dlen_* ≥ 7;

else if *dtype_a* = 1, then *dlen_* ≥ 9.

laf (local) INTEGER. The size of the array *af*.

Must be *laf* ≥ (NB+2*bw)*bw.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of size *lwork*.

lwork (local or global) INTEGER. The size of the *work* array, must be *lwork*≥bw^2.

Output Parameters

a On exit, if *info*=0, contains the permuted triangular factor *U* or *L* from the Cholesky factorization of the band matrix *A*(1:*n*, *ja*:*ja*+*n*-1), as specified by *uplo*.

af (local)

REAL for pspbtrf

DOUBLE PRECISON for pdpbtrf

COMPLEX for pcpbtrf

DOUBLE COMPLEX for pzpbtrf.

Array of size *laf*. Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?pbtrf and stored in *af*. Note that if a linear system is to be solved using p?pbtrs after the factorization routine,*af* must not be altered.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info>0:

If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?pttrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite tridiagonal distributed matrix.

Syntax

```
call pspttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pdpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pcpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pzpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
```

Include Files

Description

The p?pttrfroutine computes the Cholesky factorization of an *n*-by-*n* real symmetric or complex hermitian positive-definite tridiagonal distributed matrix *A*(1:*n*, *ja*:*ja*+*n*-1).

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

A(1:*n*, *ja*:*ja*+*n*-1) = $P*L*D*L^H*P^T$, or

A(1:*n*, *ja*:*ja*+*n*-1) = $P*U^H*D*U*P^T$,

where *P* is a permutation matrix, and *U* and *L* are tridiagonal upper and lower triangular matrices, respectively.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix *A*(1:*n*, *ja*:*ja*+*n*-1) (*n*≥ 0).
d, *e*	(local) REAL for pspttrf DOUBLE PRECISON for pdpttrf COMPLEX for pcpttrf DOUBLE COMPLEX for pzpttrf. Pointers into the local memory to arrays of size *nb_a* each. On entry, the array *d* contains the local part of the global vector storing the main diagonal of the distributed matrix *A*. On entry, the array *e* contains the local part of the global vector storing the upper diagonal of the distributed matrix *A*.
ja	(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*. If *dtype_a* = 501, then *dlen_*≥ 7; else if *dtype_a* = 1, then *dlen_*≥ 9.
laf	(local) INTEGER. The size of the array *af*. Must be *laf*≥*nb_a*+2. If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).
work	(local) Same type as *d* and *e*. Workspace array of size *lwork* .
lwork	(local or global) INTEGER. The size of the *work* array, must be at least *lwork*≥ 8*NPCOL.

Output Parameters

d, *e*	On exit, overwritten by the details of the factorization.

af	(local) REAL for pspttrf DOUBLE PRECISION for pdpttrf COMPLEX for pcpttrf DOUBLE COMPLEX for pzpttrf. Array of size *laf*. Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?pttrf and stored in *af*. Note that if a linear system is to be solved using p?pttrs after the factorization routine,*af* must not be altered.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. *info*> 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed. If *info* = *k* > NPROCS, the submatrix stored on processor *info*–NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Solving Systems of Linear Equations: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see Routines for Matrix Factorization in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

p?getrs

Solves a system of distributed linear equations with a general square matrix, using the LU factorization computed by p?getrf.

Syntax

```
call psgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

```
call pzgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

Include Files

Description

The p?getrsroutine solves a system of distributed linear equations with a general *n*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) using the *LU* factorization computed by p?getrf.

The system has one of the following forms specified by *trans*:

sub(*A*)**X* = sub(*B*) (no transpose),

sub(*A*)^T**X* = sub(*B*) (transpose),

sub(*A*)^H**X* = sub(*B*) (conjugate transpose),

where sub(*B*) = *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*nrhs*-1).

Before calling this routine,you must call p?getrf to compute the *LU* factorization of sub(*A*).

Input Parameters

trans	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', then sub(*A*)**X* = sub(*B*) is solved for *X*. If *trans* = 'T', then sub(*A*)^T**X* = sub(*B*) is solved for *X*. If *trans* = 'C', then sub(*A*)^H **X* = sub(*B*) is solved for *X*.
n	(global) INTEGER. The number of linear equations; the order of the matrix sub(*A*) (*n*≥0).
nrhs	(global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix sub(*B*) (*nrhs*≥0).
a, *b*	(local) REAL for psgetrs DOUBLE PRECISION for pdgetrs COMPLEX for pcgetrs DOUBLE COMPLEX for pzgetrs. Pointers into the local memory to arrays of local sizes (*lld_a*, *LOCc*(*ja*+*n*-1)) and (*lld_b*, *LOCc*(*jb*+*nrhs*-1)), respectively. On entry, the array *a* contains the local pieces of the factors *L* and *U* from the factorization sub(*A*) = *P***L***U*; the unit diagonal elements of *L* are not stored. On entry, the array *b* contains the right hand sides sub(*B*).
ia, *ja*	(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.
desca	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

ipiv	(local) INTEGER Array of size of *LOCr(m_a) + mb_a*. Contains the pivoting information: local row *i* of the matrix was interchanged with the global row *ipiv*(*i*). This array is tied to the distributed matrix *A*.
ib, *jb*	(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the matrix sub(*B*), respectively.
descb	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Output Parameters

b	On exit, overwritten by the solution distributed matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gbtrs

Solves a system of distributed linear equations with a general band matrix, using the LU factorization computed by p?gbtrf.

Syntax

```
call psgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf, work, lwork, info)
call pdgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf, work, lwork, info)
call pcgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf, work, lwork, info)
call pzgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf, work, lwork, info)
```

Include Files

Description

The p?gbtrs routine solves a system of distributed linear equations with a general band distributed matrix sub(*A*) = *A*(1:*n*, *ja*:*ja*+*n*-1) using the *LU* factorization computed by p?gbtrf.

The system has one of the following forms specified by *trans*:

sub(*A*)**X* = sub(*B*) (no transpose),

sub(*A*)^T*X = sub(*B*) (transpose),

sub(*A*)^H**X* = sub(*B*) (conjugate transpose),

where sub(*B*) = *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

Before calling this routine,you must call p?gbtrf to compute the *LU* factorization of sub(*A*).

Input Parameters

trans

(global) CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', then sub(*A*)**X* = sub(*B*) is solved for *X*.

If *trans* = 'T', then sub(*A*)^T**X* = sub(*B*) is solved for *X*.

If *trans* = 'C', then sub(*A*)^H **X* = sub(*B*) is solved for *X*.

n

(global) INTEGER. The number of linear equations; the order of the distributed matrix sub(*A*) (*n*≥ 0).

bwl

(global) INTEGER. The number of sub-diagonals within the band of *A*(0 ≤ *bwl* ≤ *n-1*).

bwu

(global) INTEGER. The number of super-diagonals within the band of *A*(0 ≤ *bwu* ≤ *n-1*).

nrhs

(global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix sub(*B*) (*nrhs*≥ 0).

a, *b*

(local)

REAL for psgbtrs

DOUBLE PRECISION for pdgbtrs

COMPLEX for pcgbtrs

DOUBLE COMPLEX for pzgbtrs.

Pointers into the local memory to arrays of local sizes (*lld_a*,*LOCc*(*ja*+*n*-1)) and (*lld_b*,*LOCc*(*nrhs*)), respectively.

The array *a* contains details of the *LU* factorization of the distributed band matrix *A*.

On entry, the array *b* contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

ja

(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501, then *dlen_*≥ 7;

else if *dtype_a* = 1, then *dlen_*≥ 9.

ib

(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

descb

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_b* = 502, then *dlen_*≥ 7;

	else if *dtype_b* = 1, then *dlen_* ≥ 9.
laf	(local) INTEGER. The size of the array *af*.
	Must be *laf*≥*nb_a**(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2**bwu*).
	If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).
work	(local) Same type as *a*. Workspace array of size *lwork*.
lwork	(local or global) INTEGER. The size of the *work* array, must be at least *lwork*≥*nrhs**(*nb_a*+2**bwl*+4**bwu*).

Output Parameters

ipiv	(local) INTEGER array.
	The size of *ipiv* must be ≥*nb_a*.
	Contains pivot indices for local factorizations. Note that you should not alter the contents of this array between factorization and solve.
b	On exit, overwritten by the local pieces of the solution distributed matrix *X*.
af	(local)
	REAL for psgbtrs
	DOUBLE PRECISION for pdgbtrs
	COMPLEX for pcgbtrs
	DOUBLE COMPLEX for pzgbtrs.
	Array of size *laf*.
	Auxiliary Fill-in space. The fill-in space is created in a call to the factorization routine p?gbtrf and is stored in *af*.
	Note that if a linear system is to be solved using p?gbtrs after the factorization routine,*af* must not be altered after the factorization.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info*=0, the execution is successful.
	info < 0:
	If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrs

Solves a system of linear equations with a diagonally dominant-like banded distributed matrix using the factorization computed by p?dbtrf.

Syntax

call psdbtrs(*trans*, *n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pddbtrs(*trans*, *n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pcdbtrs(*trans*, *n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pzdbtrs(*trans*, *n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

Include Files

Description

The p?dbtrsroutine solves for X one of the systems of equations:

sub(A)*X = sub(B),

(sub(A))^T*X = sub(B), or

(sub(A))^H*X = sub(B),

where sub(A) = A(1:n, ja:ja+n-1) is a diagonally dominant-like banded distributed matrix, and sub(B) denotes the distributed matrix B(ib:ib+n-1, 1:$nrhs$).

This routine uses the *LU* factorization computed by p?dbtrf.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', then sub(A)*X = sub(B) is solved for X.

If *trans* = 'T', then (sub(A))^T*X = sub(B) is solved for X.

If *trans* = 'C', then (sub(A))^H*X = sub(B) is solved for X.

n (global) INTEGER. The order of the distributed matrix sub(A) ($n \geq 0$).

bwl (global) INTEGER. The number of subdiagonals within the band of A ($0 \leq bwl \leq n-1$).

bwu (global) INTEGER. The number of superdiagonals within the band of A ($0 \leq bwu \leq n-1$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix sub(B) ($nrhs \geq 0$).

a, *b* (local)

REAL for psdbtrs

DOUBLE PRECISON for pddbtrs

COMPLEX for pcdbtrs

DOUBLE COMPLEX for pzdbtrs.

Pointers into the local memory to arrays of local sizes (*lld_a*,*LOCc*(*ja*+*n*-1)) and (*lld_b*,*LOCc*(*nrhs*)), respectively.

On entry, the array *a* contains details of the *LU* factorization of the band matrix *A*, as computed by p?dbtrf.

On entry, the array *b* contains the local pieces of the right hand side distributed matrix sub(*B*).

ja

(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501, then *dlen_* ≥ 7;

else if *dtype_a* = 1, then *dlen_* ≥ 9.

ib

(global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

If *dtype_b* = 502, then *dlen_* ≥ 7;

else if *dtype_b* = 1, then *dlen_* ≥ 9.

af, *work*

(local)

REAL for psdbtrs

DOUBLE PRECISION for pddbtrs

COMPLEX for pcdbtrs

DOUBLE COMPLEX for pzdbtrs.

Arrays of size *laf* and *lwork*, respectively The array *af* contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?dbtrf and is stored in *af*.

The array *work* is a workspace array.

laf

(local) INTEGER. The size of the array *af*.

Must be *laf*≥NB*(*bwl*+*bwu*)+6*(max(*bwl*,*bwu*))^2 .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

lwork

(local or global) INTEGER. The size of the array *work*, must be at least

lwork≥ (max(*bwl*,*bwu*))^2.

Output Parameters

b

On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dttrs

Solves a system of linear equations with a diagonally dominant-like tridiagonal distributed matrix using the factorization computed by p?dttrf.

Syntax

```
call psdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pddttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Include Files

Description

The p?dttrsroutine solves for X one of the systems of equations:

sub(A)*X = sub(B),

(sub(A))^T*X = sub(B), or

(sub(A))^H*X = sub(B),

where sub(A) =A(1:n, ja:ja+n-1) is a diagonally dominant-like tridiagonal distributed matrix, and sub(B) denotes the distributed matrix B(ib:ib+n-1, 1:$nrhs$).

This routine uses the *LU* factorization computed by p?dttrf.

Input Parameters

trans	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', then sub(A)*X = sub(B) is solved for X. If *trans* = 'T', then (sub(A))^T*X = sub(B) is solved for X. If *trans* = 'C', then (sub(A))^H*X = sub(B) is solved for X.
n	(global) INTEGER. The order of the distributed matrix sub(A) ($n \geq 0$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix sub(*B*) (*nrhs*≥ 0).

dl, *d*, *du* (local)

REAL for psdttrs

DOUBLE PRECISON for pddttrs

COMPLEX for pcdttrs

DOUBLE COMPLEX for pzdttrs.

Pointers to the local arrays of size *nb_a* each.

On entry, these arrays contain details of the factorization. Globally, *dl*(1) and *du*(*n*) are not referenced; *dl* and *du* must be aligned with *d*.

ja (global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501 or *dtype_a* = 502, then *dlen_*≥ 7;

else if *dtype_a* = 1, then *dlen_*≥ 9.

b (local) Same type as *d*.

Pointer into the local memory to an array of local size (*lld_b*,*LOCc*(*nrhs*))

On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix sub(*B*).

ib (global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

If *dtype_b* = 502, then *dlen_*≥ 7;

else if *dtype_b* = 1, then *dlen_*≥ 9.

af, *work* (local) REAL for psdttrs

DOUBLE PRECISION for pddttrs

COMPLEX for pcdttrs

DOUBLE COMPLEX for pzdttrs.

Arrays of size *laf* and (*lwork*), respectively.

The array *af* contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?dttrf and is stored in *af*. If a linear system is to be solved using p?dttrs after the factorization routine,*af* must not be altered.

The array *work* is a workspace array.

laf	(local) INTEGER. The size of the array *af*. Must be *laf*≥NB*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2**bwu*). If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).
lwork	(local or global) INTEGER. The size of the array *work*, must be at least *lwork*≥ 10*NPCOL+4**nrhs*.

Output Parameters

b	On exit, this array contains the local pieces of the solution distributed matrix *X*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?potrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian distributed positive-definite matrix.

Syntax

```
call pspotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pzpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Include Files

Description

The p?potrsroutine solves for *X* a system of distributed linear equations in the form:

sub(*A*)**X* = sub(*B*) ,

where sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) is an *n*-by-*n* real symmetric or complex Hermitian positive definite distributed matrix, and sub(*B*) denotes the distributed matrix *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*nrhs*-1).

This routine uses Cholesky factorization

sub(*A*) = U^H**U*, or sub(*A*) = *L**L^H

computed by p?potrf.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', upper triangle of sub(*A*) is stored;

If *uplo* = 'L', lower triangle of sub(*A*) is stored.

n (global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix sub(*B*) (*nrhs*≥0).

a, *b* (local)

REAL for pspotrs

DOUBLE PRECISION for pdpotrs

COMPLEX for pcpotrs

DOUBLE COMPLEX for pzpotrs.

Pointers into the local memory to arrays of local sizes

(*lld_a*,*LOCc*(*ja*+*n*-1)) and (*lld_b*,*LOCc*(*jb*+nrhs-1)), respectively.

The array *a* contains the factors *L* or *U* from the Cholesky factorization sub(*A*) = *L***L*^{H} or sub(*A*) = *U*^{H}**U*, as computed by p?potrf.

On entry, the array *b* contains the local pieces of the right hand sides sub(*B*).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

ib, *jb* (global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the matrix sub(*B*), respectively.

descb (local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Output Parameters

b Overwritten by the local pieces of the solution matrix *X*.

info INTEGER. If *info*=0, the execution is successful.

info < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian positive-definite band matrix.

Syntax

```
call pspbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pdpbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcpbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzpbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Include Files

Description

The p?pbtrsroutine solves for X a system of distributed linear equations in the form:

sub(A)*X = sub(B) ,

where sub(A) = A(1:n, ja:ja+n-1) is an n-by-n real symmetric or complex Hermitian positive definite distributed band matrix, and sub(B) denotes the distributed matrix B(ib:ib+n-1, 1:$nrhs$).

This routine uses Cholesky factorization

sub(A) = $P*U^H*U*P^T$, or sub(A) = $P*L*L^H*P^T$

computed by p?pbtrf.

Input Parameters

uplo	(global) CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', upper triangle of sub(A) is stored; If *uplo* = 'L', lower triangle of sub(A) is stored.
n	(global) INTEGER. The order of the distributed matrix sub(A) (n≥0).
bw	(global) INTEGER. The number of superdiagonals of the distributed matrix if *uplo* = 'U', or the number of subdiagonals if *uplo* = 'L' (bw≥0).
nrhs	(global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix sub(B) ($nrhs$≥0).
a, *b*	(local) REAL for pspbtrs DOUBLE PRECISION for pdpbtrs COMPLEX for pcpbtrs DOUBLE COMPLEX for pzpbtrs. Pointers into the local memory to arrays of local sizes (*lld_a*, *LOCc*(*ja*+*n*-1)) and (*lld_b*, *LOCc*(*nrhs*-1)), respectively. The array *a* contains the permuted triangular factor U or L from the Cholesky factorization sub(A) = $P*U^H*U*P^T$, or sub(A) = $P*L*L^H*P^T$ of the band matrix A, as returned by p?pbtrf.

	On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix sub(*B*).
ja	(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*. If *dtype_a* = 501, then *dlen_* ≥ 7; else if *dtype_a* = 1, then *dlen_* ≥ 9.
ib	(global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix sub(*B*).
descb	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*. If *dtype_b* = 502, then *dlen_* ≥ 7; else if *dtype_b* = 1, then *dlen_* ≥ 9.
af, *work*	(local) Arrays, same type as *a*. The array *af* is of size *laf*. It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?dbtrf and is stored in *af*. The array *work* is a workspace array of size *lwork*.
laf	(local) INTEGER. The size of the array *af*. Must be *laf*≥*nrhs*bw*. If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).
lwork	(local or global) INTEGER. The size of the array *work*, must be at least *lwork*≥bw^2.

Output Parameters

b	On exit, if *info*=0, this array contains the local pieces of the *n*-by-*nrhs* solution distributed matrix *X*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal distributed matrix using the factorization computed by `p?pttrf`.

Syntax

```
call pspttrs(n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pdpttrs(n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcpttrs(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzpttrs(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Include Files

Description

The `p?pttrs`routine solves for *X* a system of distributed linear equations in the form:

sub(*A*)**X* = sub(*B*) ,

where sub(*A*) = *A*(1:*n*, *ja*:*ja*+*n*-1) is an *n*-by-*n* real symmetric or complex Hermitian positive definite tridiagonal distributed matrix, and sub(*B*) denotes the distributed matrix *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

This routine uses the factorization

sub(*A*) = $P*L*D*L^H*P^T$, or sub(*A*) = $P*U^H*D*U*P^T$

computed by `p?pttrf`.

Input Parameters

uplo (global, used in complex flavors only)

`CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, upper triangle of sub(*A*) is stored;

If *uplo* = `'L'`, lower triangle of sub(*A*) is stored.

n (global) `INTEGER`. The order of the distributed matrix sub(*A*) (*n*≥0).

nrhs (global) `INTEGER`. The number of right hand sides; the number of columns of the distributed matrix sub(*B*) (*nrhs*≥0).

d, *e* (local)

`REAL` for `pspttrs`

`DOUBLE PRECISON` for `pdpttrs`

`COMPLEX` for `pcpttrs`

`DOUBLE COMPLEX` for `pzpttrs`.

Pointers into the local memory to arrays of size *nb_a* each.

These arrays contain details of the factorization as returned by `p?pttrf`

ja (global) `INTEGER`. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501 or *dtype_a* = 502, then *dlen_* ≥ 7;

else if *dtype_a* = 1, then *dlen_* ≥ 9.

b (local) Same type as *d*, *e*.

Pointer into the local memory to an array of local size

(*lld_b*, *LOCc*(*nrhs*)).

On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix sub(*B*).

ib (global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

If *dtype_b* = 502, then *dlen_* ≥ 7;

else if *dtype_b* = 1, then *dlen_* ≥ 9.

af, *work* (local) REAL for pspttrs

DOUBLE PRECISION for pdpttrs

COMPLEX for pcpttrs

DOUBLE COMPLEX for pzpttrs.

Arrays of size *laf* and (*lwork*), respectively. The array *af* contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?pttrf and is stored in *af*.

The array *work* is a workspace array.

laf (local) INTEGER. The size of the array *af*.

Must be *laf*≥*nb_a*+2.

If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

lwork (local or global) INTEGER. The size of the array *work*, must be at least

lwork≥ (10+2**min*(100,*nrhs*))*NPCOL+4**nrhs*.

Output Parameters

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1)) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful.

info < 0:

if the *i*-th argument is an array and the *j*-th entry had an illegal value, then `info` = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then `info` = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trtrs

Solves a system of linear equations with a triangular distributed matrix.

Syntax

```
call pstrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdtrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pctrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pztrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Include Files

Description

The `p?trtrs`routine solves for *X* one of the following systems of linear equations:

sub(*A*)**X* = sub(*B*),

(sub(*A*))^T**X* = sub(*B*), or

(sub(*A*))^H**X* = sub(*B*),

where sub(*A*) = *A*(`ia`:`ia`+*n*-1, `ja`:`ja`+*n*-1) is a triangular distributed matrix of order `n`, and sub(*B*) denotes the distributed matrix *B*(`ib`:`ib`+*n*-1, `jb`:`jb`+`nrhs`-1).

A check is made to verify that sub(*A*) is nonsingular.

Input Parameters

`uplo` (global) `CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates whether sub(*A*) is upper or lower triangular:

If `uplo = 'U'`, then sub(*A*) is upper triangular.

If `uplo = 'L'`, then sub(*A*) is lower triangular.

`trans` (global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Indicates the form of the equations:

If `trans = 'N'`, then sub(*A*)**X* = sub(*B*) is solved for *X*.

If `trans = 'T'`, then sub(*A*)^T**X* = sub(*B*) is solved for *X*.

If `trans = 'C'`, then sub(*A*)^H**X* = sub(*B*) is solved for *X*.

`diag` (global) `CHARACTER*1`. Must be `'N'` or `'U'`.

If `diag = 'N'`, then sub(*A*) is not a unit triangular matrix.

If `diag = 'U'`, then sub(*A*) is unit triangular.

`n` (global) `INTEGER`. The order of the distributed matrix sub(*A*) `(n≥0)`.

nrhs (global) INTEGER. The number of right-hand sides; i.e., the number of columns of the distributed matrix sub(*B*) (*nrhs*≥0).

a, *b* (local)

REAL for pstrtrs

DOUBLE PRECISION for pdtrtrs

COMPLEX for pctrtrs

DOUBLE COMPLEX for pztrtrs.

Pointers into the local memory to arrays of local sizes (*lld_a*, *LOCc*(*ja*+*n*-1)) and (*lld_b*, *LOCc*(*jb*+nrhs-1)), respectively.

The array *a* contains the local pieces of the distributed triangular matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix, and the strictly lower triangular part of sub(*A*) is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

If *diag* = 'U', the diagonal elements of sub(*A*) are also not referenced and are assumed to be 1.

On entry, the array *b* contains the local pieces of the right hand side distributed matrix sub(*B*).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

ib, *jb* (global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the matrix sub(*B*), respectively.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Output Parameters

b On exit, if *info*=0, sub(*B*) is overwritten by the solution matrix *X*.

info INTEGER. If *info*=0, the execution is successful.

info < 0:

if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info> 0:

if *info* = *i*, the *i*-th diagonal element of sub(*A*) is zero, indicating that the submatrix is singular and the solutions *X* have not been computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Estimating the Condition Number: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations. Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

p?gecon

Estimates the reciprocal of the condition number of a general distributed matrix in either the 1-norm or the infinity-norm.

Syntax

```
call psgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork, info)
call pdgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork, info)
call pcgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork, info)
call pzgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The p?gecon routine estimates the reciprocal of the condition number of a general distributed real/complex matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) in either the 1-norm or infinity-norm, using the *LU* factorization computed by p?getrf.

An estimate is obtained for $||(\text{sub}(A))^{-1}||$, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

norm (global) CHARACTER*1. Must be '1' or 'O' or 'I'.

Specifies whether the 1-norm condition number or the infinity-norm condition number is required.

If *norm* = '1' or 'O', then the 1-norm is used;

If *norm* = 'I', then the infinity-norm is used.

n (global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥ 0).

a (local)

REAL for psgecon

DOUBLE PRECISION for pdgecon

COMPLEX for pcgecon

DOUBLE COMPLEX for pzgecon.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)).

The array *a* contains the local pieces of the factors *L* and *U* from the factorization sub(*A*) = *P***L***U*; the unit diagonal elements of *L* are not stored.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

anorm (global) REAL for single precision flavors, DOUBLE PRECISION for double precision flavors.

If *norm* = '1' or 'O', the 1-norm of the original distributed matrix sub(*A*);

If *norm* = 'I', the infinity-norm of the original distributed matrix sub(*A*).

work (local)

REAL for psgecon

DOUBLE PRECISION for pdgecon

COMPLEX for pcgecon

DOUBLE COMPLEX for pzgecon.

The array *work* of size *lwork* is a workspace array.

lwork (local or global) INTEGER. The size of the array *work*.

For real flavors:

lwork must be at least

lwork≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))+2**LOCc*(*n*+mod(*ja*-1,*nb_a*)) +max(2, max(*nb_a**max(1, iceil(NPROW-1, NPCOL)),*LOCc*(*n* +mod(*ja*-1,*nb_a*)) + *nb_a**max(1, iceil(NPCOL-1, NPROW)))).

For complex flavors:

lwork must be at least

lwork≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))+max(2, max(*nb_a**iceil(NPROW-1, NPCOL),*LOCc*(*n*+mod(*ja*-1,*nb_a*))+ *nb_a**iceil(NPCOL-1, NPROW))).

LOCr and *LOCc* values can be computed using the ScaLAPACK tool function numroc; NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

NOTE
`iceil(x,y)` is the ceiling of x/y, and `mod(x,y)` is the integer remainder of x/y.

iwork (local) INTEGER. Workspace array of size *liwork*. Used in real flavors only.

liwork (local or global) INTEGER. The size of the array *iwork*; used in real flavors only. Must be at least

liwork≥*LOCr*(*n*+mod(*ia*-1,*mb_a*)).

rwork (local) REAL for pcgecon

DOUBLE PRECISION for pzgecon

Workspace array of size *lrwork*. Used in complex flavors only.

lrwork (local or global) INTEGER. The size of the array *rwork*; used in complex flavors only. Must be at least

lrwork≥ max(1, 2**LOCc*(*n*+mod(*ja*-1,*nb_a*))).

Output Parameters

rcond (global) REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The reciprocal of the condition number of the distributed matrix sub(*A*). See Description.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?pocon

Estimates the reciprocal of the condition number (in the 1 - norm) of a symmetric / Hermitian positive-definite distributed matrix.

Syntax

```
call pspocon(uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork, info)
```

```
call pdpocon(uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork, info)
call pcpocon(uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork, info)
call pzpocon(uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The p?poconroutine estimates the reciprocal of the condition number (in the 1 - norm) of a real symmetric or complex Hermitian positive definite distributed matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), using the Cholesky factorization sub(*A*) = U^H*U or sub(*A*) = $L*L^H$ computed by p?potrf.

An estimate is obtained for $||(\text{sub}(A))^{-1}||$, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the factor stored in sub(*A*) is upper or lower triangular.

If *uplo* = 'U', sub(*A*) stores the upper triangular factor *U* of the Cholesky factorization sub(*A*) = U^H*U.

If *uplo* = 'L', sub(*A*) stores the lower triangular factor *L* of the Cholesky factorization sub(*A*) = $L*L^H$.

n (global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).

a (local)

REAL for pspocon

DOUBLE PRECISION for pdpocon

COMPLEX for pcpocon

DOUBLE COMPLEX for pzpocon.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)).

The array *a* contains the local pieces of the factors *L* or *U* from the Cholesky factorization sub(*A*) = U^H*U, or sub(*A*) = $L*L^H$, as computed by p?potrf.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

anorm (global) REAL for single precision flavors,

DOUBLE PRECISION for double precision flavors.

The 1-norm of the symmetric/Hermitian distributed matrix sub(*A*).

work (local)

`REAL` for `pspocon`

`DOUBLE PRECISION` for `pdpocon`

`COMPLEX` for `pcpocon`

`DOUBLE COMPLEX` for `pzpocon`.

The array *work* of size *lwork* is a workspace array.

lwork (local or global) `INTEGER`. The size of the array *work*.

For real flavors:

lwork must be at least

lwork≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))+2**LOCc*(*n*+mod(*ja*-1,*nb_a*)) +max(2, max(*nb_a**iceil(NPROW-1, NPCOL),*LOCc*(*n* +mod(*ja*-1,*nb_a*))+*nb_a**iceil(NPCOL-1, NPROW))).

For complex flavors:

lwork must be at least

lwork≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))+max(2, max(*nb_a**max(1,iceil(NPROW-1, NPCOL)),*LOCc*(*n*+mod(*ja*-1,*nb_a*)) +*nb_a**max(1,iceil(NPCOL-1, NPROW)))).

If *lwork* = -1, then *lwork* is a global input and a workspace query is assumed. The routine only calculates the minimum and optimal size for all work arrays. Each value is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

NOTE
`iceil(x,y)` is the ceiling of *x*/*y*, and `mod(x,y)` is the integer remainder of *x*/*y*.

iwork (local) `INTEGER`. Workspace array of size *liwork*. Used in real flavors only.

liwork (local or global) `INTEGER`. The size of the array *iwork*; used in real flavors only. Must be at least *liwork*≥*LOCr*(*n*+mod(*ia*-1,*mb_a*)).

If *liwork* = -1, then *liwork* is a global input and a workspace query is assumed. The routine only calculates the minimum and optimal size for all work arrays. Each value is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

rwork (local) `REAL` for `pcpocon`

`DOUBLE PRECISION` for `pzpocon`

Workspace array of size *lrwork*. Used in complex flavors only.

lrwork (local or global) `INTEGER`. The size of the array *rwork*; used in complex flavors only. Must be at least *lrwork*≥ 2**LOCc*(*n*+mod(*ja*-1,*nb_a*)).

If *lrwork* = -1, then *lrwork* is a global input and a workspace query is assumed. The routine only calculates the minimum and optimal size for all work arrays. Each value is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

rcond

(global) REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The reciprocal of the condition number of the distributed matrix sub(*A*).

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1)

On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1)

On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info

(global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trcon

Estimates the reciprocal of the condition number of a triangular distributed matrix in either 1-norm or infinity-norm.

Syntax

```
call pstrcon(norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork, iwork, liwork, info)
call pdtrcon(norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork, iwork, liwork, info)
call pctrcon(norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork, rwork, lrwork, info)
call pztrcon(norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The p?trconroutine estimates the reciprocal of the condition number of a triangular distributed matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), in either the 1-norm or the infinity-norm.

The norm of sub(*A*) is computed and an estimate is obtained for $||(\text{sub}(A))^{-1}||$, then the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

norm (global) CHARACTER*1. Must be '1' or 'O' or 'I'.

Specifies whether the 1-norm condition number or the infinity-norm condition number is required.

If *norm* = '1' or 'O', then the 1-norm is used;

If *norm* = 'I', then the infinity-norm is used.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', sub(*A*) is upper triangular. If *uplo* = 'L', sub(*A*) is lower triangular.

diag (global) CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', sub(*A*) is non-unit triangular. If *diag* = 'U', sub(*A*) is unit triangular.

n (global) INTEGER. The order of the distributed matrix sub(*A*), (*n*≥0).

a (local)

REAL for pstrcon

DOUBLE PRECISION for pdtrcon

COMPLEX for pctrcon

DOUBLE COMPLEX for pztrcon.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)).

The array *a* contains the local pieces of the triangular distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of this distributed matrix contains the upper triangular matrix, and its strictly lower triangular part is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of this distributed matrix contains the lower triangular matrix, and its strictly upper triangular part is not referenced.

If *diag* = 'U', the diagonal elements of sub(*A*) are also not referenced and are assumed to be 1.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work (local)

REAL for pstrcon

DOUBLE PRECISION for pdtrcon

COMPLEX for pctrcon

DOUBLE COMPLEX for pztrcon.

The array *work* of size *lwork* is a workspace array.

lwork (local or global) INTEGER. The size of the array *work*.

For real flavors:

lwork must be at least

lwork≥ 2*LOCr(*n*+mod(*ia*-1,*mb_a*))+LOCc(*n*+mod(*ja*-1,*nb_a*))
+max(2, max(*nb_a*max(1,iceil(NPROW-1, NPCOL)),

LOCc(*n*+mod(*ja*-1,*nb_a*))+*nb_a*max(1,iceil(NPCOL-1, NPROW)))).

For complex flavors:

lwork must be at least

lwork≥ 2*LOCr(*n*+mod(*ia*-1,*mb_a*))+max(2,
max(*nb_a*iceil(NPROW-1, NPCOL),

LOCc(*n*+mod(*ja*-1,*nb_a*))+*nb_a*iceil(NPCOL-1, NPROW))).

NOTE
iceil(*x*,*y*) is the ceiling of *x*/*y*, and mod(*x*,*y*) is the integer remainder of *x*/*y*.

iwork (local) INTEGER. Workspace array of size *liwork*. Used in real flavors only.

liwork (local or global) INTEGER. The size of the array *iwork*; used in real flavors only. Must be at least

liwork≥LOCr(*n*+mod(*ia*-1,*mb_a*)).

rwork (local) REAL for pcpocon

DOUBLE PRECISION for pzpocon

Workspace array of size *lrwork*. Used in complex flavors only.

lrwork (local or global) INTEGER. The size of the array *rwork*; used in complex flavors only. Must be at least

lrwork≥LOCc(*n*+mod(*ja*-1,*nb_a*)).

Output Parameters

rcond (global) REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

	The reciprocal of the condition number of the distributed matrix sub(*A*).
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
iwork(1)	On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).
rwork(1)	On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Refining the Solution and Estimating Its Error: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see Routines for Matrix Factorization and Solving Systems of Linear Equations).

p?gerfs

Improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

Syntax

```
call psgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pdgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pcgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

```
call pzgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The p?gerfs routine improves the computed solution to one of the systems of linear equations

sub(*A*)*sub(*X*) = sub(*B*),

sub(*A*)^T*sub(*X*) = sub(*B*), or

sub(*A*)^H*sub(*X*) = sub(*B*) and provides error bounds and backward error estimates for the solution.

Here sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), sub(*B*) = *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*nrhs*-1), and sub(*X*) = *X*(*ix*:*ix*+*n*-1, *jx*:*jx*+*nrhs*-1).

Input Parameters

trans

(global) CHARACTER*1. Must be 'N' or 'T' or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form sub(*A*)*sub(*X*) = sub(*B*) (No transpose);

If *trans* = 'T', the system has the form sub(*A*)^T*sub(*X*) = sub(*B*) (Transpose);

If *trans* = 'C', the system has the form sub(*A*)^H*sub(*X*) = sub(*B*) (Conjugate transpose).

n

(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥ 0).

nrhs

(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices sub(*B*) and sub(*X*) (*nrhs*≥ 0).

a, *af*, *b*, *x*

(local)

REAL for psgerfs

DOUBLE PRECISION for pdgerfs

COMPLEX for pcgerfs

DOUBLE COMPLEX for pzgerfs.

Pointers into the local memory to arrays of local sizes *a*(*lld_a*, *LOCc*(*ja*+*n*-1)), *af*(*lld_af*,*LOCc*(*jaf*+*n*-1)), *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-1)), and *x*(*lld_x*,*LOCc*(*jx*+*nrhs*-1)), respectively.

The array *a* contains the local pieces of the distributed matrix sub(*A*).

The array *af* contains the local pieces of the distributed factors of the matrix sub(*A*) = *P*L*U* as computed by p?getrf.

The array *b* contains the local pieces of the distributed matrix of right hand sides sub(*B*).

On entry, the array *x* contains the local pieces of the distributed solution matrix sub(*X*).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

iaf, *jaf*

(global) INTEGER. The row and column indices in the global matrix *AF* indicating the first row and the first column of the matrix sub(*AF*), respectively.

descaf

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *AF*.

ib, *jb*

(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the matrix sub(*B*), respectively.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

ix, *jx* (global) INTEGER. The row and column indices in the global matrix *X* indicating the first row and the first column of the matrix sub(*X*), respectively.

descx (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *X*.

ipiv (local) INTEGER.

Array of size *LOCr*(*m_af*) + *mb_af*.

This array contains pivoting information as computed by p?getrf. If *ipiv*(*i*)=*j*, then the local row *i* was swapped with the global row *j*.

This array is tied to the distributed matrix *A*.

work (local)

REAL for psgerfs

DOUBLE PRECISION for pdgerfs

COMPLEX for pcgerfs

DOUBLE COMPLEX for pzgerfs.

The array *work* of size *lwork* is a workspace array.

lwork (local or global) INTEGER. The size of the array *work*.

For real flavors:

lwork must be at least

lwork≥ 3**LOCr*(*n*+mod(*ia*-1,*mb_a*))

For complex flavors:

lwork must be at least

lwork≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

iwork (local) INTEGER. Workspace array, size *liwork*. Used in real flavors only.

liwork (local or global) INTEGER. The size of the array *iwork*; used in real flavors only. Must be at least

liwork≥*LOCr*(*n*+mod(*ib*-1,*mb_b*)).

rwork (local) REAL for pcgerfs

DOUBLE PRECISION for pzgerfs

Workspace array, size *lrwork*. Used in complex flavors only.

lrwork (local or global) INTEGER. The size of the array *rwork*; used in complex flavors only. Must be at least *lrwork*≥*LOCr*(*n*+mod(*ib*-1,*mb_b*))).

Output Parameters

x	On exit, contains the improved solution vectors.
ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays of size *LOCc*(*jb*+*nrhs*-1) each. The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*). If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error. This array is tied to the distributed matrix *X*. The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
iwork(1)	On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).
rwork(1)	On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?porfs

Improves the computed solution to a system of linear equations with symmetric/Hermitian positive definite distributed matrix and provides error bounds and backward error estimates for the solution.

Syntax

```
call psporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib, jb, descb, x,
ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pdporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib, jb, descb, x,
ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pcporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib, jb, descb, x,
ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

```
call pzporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib, jb, descb, x,
ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The p?porfsroutine improves the computed solution to the system of linear equations

sub(*A*)*sub(*X*) = sub(*B*),

where sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) is a real symmetric or complex Hermitian positive definite distributed matrix and

sub(*B*) = *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*nrhs*-1),

sub(*X*) = *X*(*ix*:*ix*+*n*-1, *jx*:*jx*+*nrhs*-1)

are right-hand side and solution submatrices, respectively. This routine also provides error bounds and backward error estimates for the solution.

Input Parameters

uplo

(global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(*A*) is stored.

If *uplo* = 'U', sub(*A*) is upper triangular. If *uplo* = 'L', sub(*A*) is lower triangular.

n

(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).

nrhs

(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices sub(*B*) and sub(*X*) (*nrhs*≥0).

a, *af*, *b*, *x*

(local)

REAL for psporfs

DOUBLE PRECISION for pdporfs

COMPLEX for pcporfs

DOUBLE COMPLEX for pzporfs.

Pointers into the local memory to arrays of local sizes

a(*lld_a*, *LOCc*(*ja*+*n*-1)), *af*(*lld_af*,*LOCc*(*jaf*+*n*-1)), *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-1)), and *x*(*lld_x*,*LOCc*(*jx*+*nrhs*-1)), respectively.

The array *a* contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.

The array *af* contains the factors *L* or *U* from the Cholesky factorization sub(*A*) = *L***L*^{H} or sub(*A*) = *U*^{H}**U*, as computed by p?potrf.

On entry, the array *b* contains the local pieces of the distributed matrix of right hand sides sub(*B*).

On entry, the array *x* contains the local pieces of the solution vectors sub(*X*).

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

iaf, *jaf* — (global) INTEGER. The row and column indices in the global matrix *AF* indicating the first row and the first column of the matrix sub(*AF*), respectively.

descaf — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *AF*.

ib, *jb* — (global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the matrix sub(*B*), respectively.

descb — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

ix, *jx* — (global) INTEGER. The row and column indices in the global matrix *X* indicating the first row and the first column of the matrix sub(*X*), respectively.

descx — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *X*.

work — (local)

REAL for psporfs

DOUBLE PRECISION for pdporfs

COMPLEX for pcporfs

DOUBLE COMPLEX for pzporfs.

The array *work* of size *lwork* is a workspace array.

lwork — (local) INTEGER. The size of the array *work*.

For real flavors:

lwork must be at least

lwork≥ 3**LOCr*(*n*+mod(*ia*-1,*mb_a*))

For complex flavors:

lwork must be at least

lwork≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))

NOTE
`mod(x,y)` is the integer remainder of x/y.

iwork (local) INTEGER. Workspace array of size *liwork*. Used in real flavors only.

liwork (local or global) INTEGER. The size of the array *iwork*; used in real flavors only. Must be at least

liwork≥*LOCr*(*n*+mod(*ib*-1,*mb_b*)).

rwork (local) REAL for pcporfs

DOUBLE PRECISION for pzporfs

Workspace array of size *lrwork*. Used in complex flavors only.

lrwork (local or global) INTEGER. The size of the array *rwork*; used in complex flavors only. Must be at least *lrwork*≥*LOCr*(*n*+mod(*ib*-1,*mb_b*))).

Output Parameters

x On exit, contains the improved solution vectors.

ferr, *berr* REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays of size *LOCc*(*jb*+*nrhs*-1) each.

The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*).

If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error.

This array is tied to the distributed matrix *X*.

The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then `info` = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then `info` = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trrfs

Provides error bounds and backward error estimates for the solution to a system of linear equations with a distributed triangular coefficient matrix.

Syntax

```
call pstrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
call pdtrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
call pctrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
call pztrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The `p?trrfs`routine provides error bounds and backward error estimates for the solution to one of the systems of linear equations

sub(*A*)*sub(*X*) = sub(*B*),

sub(*A*)^T*sub(*X*) = sub(*B*), or

sub(*A*)^H*sub(*X*) = sub(*B*) ,

where sub(*A*) = *A*(`ia`:`ia`+*n*-1, `ja`:`ja`+*n*-1) is a triangular matrix,

sub(*B*) = *B*(`ib`:`ib`+*n*-1, `jb`:`jb`+`nrhs`-1), and

sub(*X*) = *X*(`ix`:`ix`+*n*-1, `jx`:`jx`+`nrhs`-1).

The solution matrix *X* must be computed by `p?trtrs` or some other means before entering this routine. The routine `p?trrfs` does not do iterative refinement because doing so cannot improve the backward error.

Input Parameters

`uplo`

(global) `CHARACTER*1`. Must be `'U'` or `'L'`.

If `uplo = 'U'`, sub(*A*) is upper triangular. If `uplo = 'L'`, sub(*A*) is lower triangular.

`trans`

(global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Specifies the form of the system of equations:

If `trans = 'N'`, the system has the form sub(*A*)*sub(*X*) = sub(*B*) (No transpose);

If *trans* = 'T', the system has the form sub(*A*)^T*sub(*X*) = sub(*B*) (Transpose);

If *trans* = 'C', the system has the form sub(*A*)^H*sub(*X*) = sub(*B*) (Conjugate transpose).

diag

CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then sub(*A*) is non-unit triangular.

If *diag* = 'U', then sub(*A*) is unit triangular.

n

(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).

nrhs

(global) INTEGER. The number of right-hand sides, that is, the number of columns of the matrices sub(*B*) and sub(*X*) (*nrhs*≥0).

a, *b*, *x*

(local)

REAL for pstrrfs

DOUBLE PRECISION for pdtrrfs

COMPLEX for pctrrfs

DOUBLE COMPLEX for pztrrfs.

Pointers into the local memory to arrays of local sizes *a*(*lld_a*, *LOCc*(*ja*+*n*-1)), *b*(*lld_b*, *LOCc*(*jb*+*nrhs*-1)), and *x*(*lld_x*, *LOCc*(*jx*+*nrhs*-1)), respectively.

The array *a* contains the local pieces of the original triangular distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.

If *diag* = 'U', the diagonal elements of sub(*A*) are also not referenced and are assumed to be 1.

On entry, the array *b* contains the local pieces of the distributed matrix of right hand sides sub(*B*).

On entry, the array *x* contains the local pieces of the solution vectors sub(*X*).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

ib, *jb*

(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the matrix sub(*B*), respectively.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

ix, *jx* (global) INTEGER. The row and column indices in the global matrix *X* indicating the first row and the first column of the matrix sub(*X*), respectively.

descx (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *X*.

work (local)

REAL for pstrrfs

DOUBLE PRECISION for pdtrrfs

COMPLEX for pctrrfs

DOUBLE COMPLEX for pztrrfs.

The array *work* of size *lwork* is a workspace array.

lwork (local) INTEGER. The size of the array *work*.

For real flavors:

lwork must be at least *lwork*≥ 3**LOCr*(*n*+mod(*ia*-1,*mb_a*))

For complex flavors:

lwork must be at least

lwork≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

iwork (local) INTEGER. Workspace array of size *liwork*. Used in real flavors only.

liwork (local or global) INTEGER. The size of the array *iwork*; used in real flavors only. Must be at least

liwork≥*LOCr*(*n*+mod(*ib*-1,*mb_b*)).

rwork (local) REAL for pctrrfs

DOUBLE PRECISION for pztrrfs

Workspace array of size *lrwork*. Used in complex flavors only.

lrwork (local or global) INTEGER. The size of the array *rwork*; used in complex flavors only. Must be at least *lrwork*≥*LOCr*(*n*+mod(*ib*-1,*mb_b*))).

Output Parameters

ferr, *berr* REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays of size *LOCc*(*jb*+*nrhs*-1) each.

The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*).

If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error.

This array is tied to the distributed matrix *X*.

The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Matrix Inversion: ScaLAPACK Computational Routines

This sections describes ScaLAPACK routines that compute the inverse of a matrix based on the previously obtained factorization. Note that it is not recommended to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see Solving Systems of Linear Equations); this is more efficient and more accurate.

p?getri

Computes the inverse of a LU-factored distributed matrix.

Syntax

```
call psgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pdgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pcgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pzgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
```

Include Files

Description

The p?getriroutine computes the inverse of a general distributed matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja* +*n*-1) using the *LU* factorization computed by p?getrf. This method inverts *U* and then computes the inverse of sub(*A*) by solving the system

inv(sub(*A*))**L* = *inv*(*U*)

for *inv*(sub(*A*)).

Input Parameters

n

(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(*A*) (*n*≥0).

a

(local)

REAL for psgetri

DOUBLE PRECISION for pdgetri

COMPLEX for pcgetri

DOUBLE COMPLEX for pzgetri.

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja* +*n*-1)).

On entry, the array *a* contains the local pieces of the *L* and *U* obtained by the factorization sub(*A*) = *P*L*U* computed by p?getrf.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work

(local)

REAL for psgetri

DOUBLE PRECISION for pdgetri

COMPLEX for pcgetri

DOUBLE COMPLEX for pzgetri.

The array *work* of size *lwork* is a workspace array.

lwork

(local) INTEGER. The size of the array *work*. *lwork* must be at least

lwork≥*LOCr*(*n*+mod(*ia*-1,*mb_a*))**nb_a*.

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

The array *work* is used to keep at most an entire column block of sub(*A*).

iwork

(local) INTEGER. Workspace array used for physically transposing the pivots, size *liwork*.

liwork

(local or global) INTEGER. The size of the array *iwork*.

The minimal value *liwork* of is determined by the following code:

```
if NPROW == NPCOL then
liwork = LOCc(n_a + mod(ja-1,nb_a))+ nb_a
else
liwork = LOCc(n_a + mod(ja-1,nb_a)) +
max(ceil(ceil(LOCr(m_a)/mb_a)/(lcm/NPROW)),nb_a)
end if
```

where lcm is the least common multiple of process rows and columns (NPROW and NPCOL).

Output Parameters

ipiv

(local) INTEGER.

Array of size *LOCr*(*m_a*)+ *mb_a*.

This array contains the pivoting information.

If *ipiv*(*i*)=*j*, then the local row *i* was swapped with the global row *j*.

This array is tied to the distributed matrix *A*.

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1)

On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance.

info

(global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info> 0:

If *info* = *i*, *U*(*i*,*i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?potri

Computes the inverse of a symmetric/Hermitian positive definite distributed matrix.

Syntax

```
call pspotri(uplo, n, a, ia, ja, desca, info)
call pdpotri(uplo, n, a, ia, ja, desca, info)
```

```
call pcpotri(uplo, n, a, ia, ja, desca, info)
call pzpotri(uplo, n, a, ia, ja, desca, info)
```

Include Files

Description

The p?potriroutine computes the inverse of a real symmetric or complex Hermitian positive definite distributed matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) using the Cholesky factorization sub(*A*) = U^H*U or sub(*A*) = $L*L^H$ computed by p?potrf.

Input Parameters

uplo — (global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(*A*) is stored.

If *uplo* = 'U', upper triangle of sub(*A*) is stored. If *uplo* = 'L', lower triangle of sub(*A*) is stored.

n — (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(*A*) (*n*≥0).

a — (local)

REAL for pspotri

DOUBLE PRECISION for pdpotri

COMPLEX for pcpotri

DOUBLE COMPLEX for pzpotri.

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja*+*n*-1)).

On entry, the array *a* contains the local pieces of the triangular factor *U* or *L* from the Cholesky factorization sub(*A*) = U^H*U, or sub(*A*) = $L*L^H$, as computed by p?potrf.

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a — On exit, overwritten by the local pieces of the upper or lower triangle of the (symmetric/Hermitian) inverse of sub(*A*).

info — (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info> 0:

If *info* = *i*, the element (*i*, *i*) of the factor *U* or *L* is zero, and the inverse could not be computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trtri

Computes the inverse of a triangular distributed matrix.

Syntax

```
call pstrtri(uplo, diag, n, a, ia, ja, desca, info)
call pdtrtri(uplo, diag, n, a, ia, ja, desca, info)
call pctrtri(uplo, diag, n, a, ia, ja, desca, info)
call pztrtri(uplo, diag, n, a, ia, ja, desca, info)
```

Include Files

Description

The p?trtriroutine computes the inverse of a real or complex upper or lower triangular distributed matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

Input Parameters

uplo

(global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the distributed matrix sub(*A*) is upper or lower triangular.

If *uplo* = 'U', sub(*A*) is upper triangular.

If *uplo* = 'L', sub(*A*) is lower triangular.

diag

CHARACTER*1. Must be 'N' or 'U'.

Specifies whether or not the distributed matrix sub(*A*) is unit triangular.

If *diag* = 'N', then sub(*A*) is non-unit triangular.

If *diag* = 'U', then sub(*A*) is unit triangular.

n

(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(*A*) (*n*≥0).

a

(local)

REAL for pstrtri

DOUBLE PRECISION for pdtrtri

COMPLEX for pctrtri

DOUBLE COMPLEX for pztrtri.

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja*+*n*-1)).

The array *a* contains the local pieces of the triangular distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix to be inverted, and the strictly lower triangular part of sub(*A*) is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a — On exit, overwritten by the (triangular) inverse of the original matrix.

info — (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info> 0:

If *info* = *k*, *A*(*ia*+*k*-1, *ja*+*k*-1) is exactly zero. The triangular matrix sub(*A*) is singular and its inverse cannot be computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Matrix Equilibration: ScaLAPACK Computational Routines

ScaLAPACK routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

p?geequ

Computes row and column scaling factors intended to equilibrate a general rectangular distributed matrix and reduce its condition number.

Syntax

```
call psgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pdgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pcgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pzgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
```

Include Files

Description

The p?geequroutine computes row and column scalings intended to equilibrate an *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) and reduce its condition number. The output array *r* returns the row scale factors r_i , and the array *c* returns the column scale factors c_j . These factors are chosen to try to make the largest element in each row and column of the matrix *B* with elements $b_{ij}=r_i*a_{ij}*c_j$ have absolute value 1.

r_i and c_j are restricted to be between *SMLNUM* = smallest safe number and *BIGNUM* = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of sub(*A*) but works well in practice.

SMLNUM and *BIGNUM* are parameters representing machine precision. You can use the ?lamch routines to compute them. For example, compute single precision values of *SMLNUM* and *BIGNUM* as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

The auxiliary function p?laqge uses scaling factors computed by p?geequ to scale a general rectangular matrix.

Input Parameters

m
(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed matrix sub(*A*) (*m*≥ 0).

n
(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed matrix sub(*A*) (*n*≥ 0).

a
(local)

REAL for psgeequ

DOUBLE PRECISION for pdgeequ

COMPLEX for pcgeequ

DOUBLE COMPLEX for pzgeequ .

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja*+*n*-1)).

The array *a* contains the local pieces of the *m*-by-*n* distributed matrix whose equilibration factors are to be computed.

ia, *ja*
(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca
(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

r, *c*
(local) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Arrays of sizes *LOCr*(*m_a*) and *LOCc*(*n_a*), respectively.

If *info* = 0, or *info*>*ia*+*m*-1, the array *r*(*ia*:*ia*+*m*-1) contains the row scale factors for sub(*A*). *r* is aligned with the distributed matrix *A*, and replicated across every process column. *r* is tied to the distributed matrix *A*.

If *info* = 0, the array *c* (*ja*:*ja*+*n*-1) contains the column scale factors for sub(*A*). *c* is aligned with the distributed matrix *A*, and replicated down every process row. *c* is tied to the distributed matrix *A*.

rowcnd, *colcnd*

(global) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

If *info* = 0 or *info*>*ia*+*m*-1, *rowcnd* contains the ratio of the smallest *r*(*i*) to the largest *r*(*i*) (i*a* ≤ *i* ≤ *ia*+*m*-1). If *rowcnd*≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r*(*ia*:*ia*+*m*-1).

If *info* = 0, *colcnd* contains the ratio of the smallest *c*(*j*) to the largest *c*(*j*) (*ja* ≤ *j* ≤ *ja*+*n*-1).

If *colcnd*≥ 0.1, it is not worth scaling by *c*(*ja*:*ja*+*n*-1).

amax

(global) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info

(global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info> 0:

If *info* = *i* and

i ≤ *m*, the *i*-th row of the distributed matrix

sub(*A*) is exactly zero;

i>*m*, the (*i* - *m*)-th column of the distributed

matrix sub(*A*) is exactly zero.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite distributed matrix and reduce its condition number.

Syntax

```
call pspoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pdpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pcpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
```

```
call pzpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
```

Include Files

Description

The p?poequ routine computes row and column scalings intended to equilibrate a real symmetric or complex Hermitian positive definite distributed matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and reduce its condition number (with respect to the two-norm). The output arrays *sr* and *sc* return the row and column scale factors

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled distributed matrix *B* with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has ones on the diagonal.

This choice of *sr* and *sc* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

The auxiliary function p?laqsy uses scaling factors computed by p?geequ to scale a general rectangular matrix.

Input Parameters

n

(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(*A*) (*n*≥0).

a

(local)

REAL for pspoequ

DOUBLE PRECISION for pdpoequ

COMPLEX for pcpoequ

DOUBLE COMPLEX for pzpoequ.

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja*+*n*-1)).

The array *a* contains the *n*-by-*n* symmetric/Hermitian positive definite distributed matrix sub(*A*) whose scaling factors are to be computed. Only the diagonal elements of sub(*A*) are referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

sr, *sc*

(local)

REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Arrays of sizes *LOCr(m_a)* and *LOCc(n_a)*, respectively.

If *info* = 0, the array *sr(ia:ia+n-1)* contains the row scale factors for sub(*A*). *sr* is aligned with the distributed matrix *A*, and replicated across every process column. *sr* is tied to the distributed matrix *A*.

If *info* = 0, the array *sc(ja:ja+n-1)* contains the column scale factors for sub(*A*). *sc* is aligned with the distributed matrix *A*, and replicated down every process row. *sc* is tied to the distributed matrix *A*.

scond (global)

REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

If *info* = 0, *scond* contains the ratio of the smallest *sr*(*i*) (or *sc*(*j*)) to the largest *sr*(*i*) (or *sc*(*j*)), with

ia≤i≤*ia+n-1* and *ja*≤j≤*ja+n-1*.

If *scond*≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *sr* (or *sc*).

amax (global)

REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info (global) INTEGER.

If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info> 0:

If *info* = *k*, the *k*-th diagonal entry of sub(*A*) is nonpositive.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Orthogonal Factorizations: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for the *QR(RQ)* and *LQ(QL)* factorization of matrices. Routines for the *RZ* factorization as well as for generalized *QR* and *RQ* factorizations are also included. For the mathematical definition of the factorizations, see the respective LAPACK sections or refer to [SLUG].

Table "Computational Routines for Orthogonal Factorizations" lists ScaLAPACK routines that perform orthogonal factorization of matrices.

Computational Routines for Orthogonal Factorizations

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	p?geqrf	p?geqpf	p?orgqr p?ungqr	p?ormqr p?unmqr
general matrices, RQ factorization	p?gerqf		p?orgrq p?ungrq	p?ormrq p?unmrq
general matrices, LQ factorization	p?gelqf		p?orglq p?unglq	p?ormlq p?unmlq
general matrices, QL factorization	p?geqlf		p?orgql p?ungql	p?ormql p?unmql
trapezoidal matrices, RZ factorization	p?tzrzf			p?ormrz p?unmrz
pair of matrices, generalized QR factorization	p?ggqrf			
pair of matrices, generalized RQ factorization	p?ggrqf			

p?geqrf

Computes the QR factorization of a general m-by-n matrix.

Syntax

```
call psgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The `p?geqrf` routine forms the *QR* factorization of a general *m*-by-*n* distributed matrix sub(*A*)= *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) as

A=*Q***R*.

Input Parameters

m (global) `INTEGER`. The number of rows in the distributed matrix sub(*A*); (*m*≥ 0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*A*); (*n*≥ 0).

a (local)

REAL for psgeqrf

DOUBLE PRECISION for pdgeqrf

COMPLEX for pcgeqrf

DOUBLE COMPLEX for pzgeqrf.

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*

work (local).

REAL for psgeqrf

DOUBLE PRECISION for pdgeqrf.

COMPLEX for pcgeqrf.

DOUBLE COMPLEX for pzgeqrf

Workspace array of size *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least *lwork*≥*nb_a* * (*mp*0+*nq*0+*nb_a*), where

iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL), and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	The elements on and above the diagonal of sub(*A*) contain the min(*m*,*n*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m*≥*n*); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau	(local)
	`REAL` for `psgeqrf`
	`DOUBLE PRECISION` for `pdgeqrf`
	`COMPLEX` for `pcgeqrf`
	`DOUBLE COMPLEX` for `pzgeqrf`.
	Array of size *LOCc*(*ja*+min(*m*,*n*)−1).
	Contains the scalar factor of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) `INTEGER`. = 0, the execution is successful. < 0, if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ja*)**H*(*ja*+*1*)*...**H*(*ja*+*k*-1),

where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau***v***v'*

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+*1*:*m*) is stored on exit in *A*(*ia*+*i*:*ia*+m-1, *ja*+*i*-1), and *tau* in *tau*(*ja*+*i*-1).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?geqpf

Computes the QR factorization of a general m-by-n matrix with pivoting.

Syntax

```
call psgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pdgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pcgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, rwork, lrwork, info)
call pzgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The p?geqpf routine forms the *QR* factorization with column pivoting of a general *m*-by-*n* distributed matrix sub(*A*)= *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) as

sub(*A*)*P*=*Q*R.

Input Parameters

m (global) INTEGER. The number of rows in the matrix sub(*A*) (*m*≥ 0).

n (global) INTEGER. The number of columns in the matrix sub(*A*) (*n*≥ 0).

a (local)

REAL for psgeqpf

DOUBLE PRECISION for pdgeqpf

COMPLEX for pcgeqpf

DOUBLE COMPLEX for pzgeqpf.

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work (local).

REAL for psgeqpf

DOUBLE PRECISION for pdgeqpf.

COMPLEX for pcgeqpf.

DOUBLE COMPLEX for pzgeqpf

Workspace array of size *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least

For real flavors:

lwork≥max(3,*mp*0+*nq*0) + *LOCc* (*ja*+*n*-*1*) + *nq*0.

For complex flavors:

lwork≥max(3,*mp*0+*nq*0) .

Here

iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL),

LOCc (*ja*+*n*-1) = numroc(*ja*+*n*-1, *nb_a*, MYCOL,*csrc_a*, NPCOL),
and numroc, indxg2p are ScaLAPACK tool functions.

You can determine MYROW, MYCOL, NPROW and NPCOL by calling the blacs_gridinfosubroutine.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

rwork (local).

REAL for pcgeqpf.

DOUBLE PRECISION for pzgeqpf.

Workspace array of size *lrwork* (complex flavors only).

lrwork (local or global) INTEGER, size of *rwork* (complex flavors only). The value of *lrwork* must be at least

lwork≥*LOCc* (*ja*+*n*-1) + *nq*0 .

Here

iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL),

LOCc (*ja*+*n*-1) = numroc(*ja*+*n*-1, *nb_a*, MYCOL,*csrc_a*, NPCOL),
and numroc, indxg2p are ScaLAPACK tool functions.

You can determine MYROW, MYCOL, NPROW and NPCOL by calling the blacs_gridinfosubroutine.

If *lrwork* = -1, then *lrwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a The elements on and above the diagonal of sub(*A*)contain the min(*m*,*n*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m*≥*n*); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

ipiv (local) INTEGER. Array of size *LOCc*(*ja*+*n*-1).

ipiv(*i*) = *k*, the local *i*-th column of sub(*A*)**P* was the global *k*-th column of sub(*A*). *ipiv* is tied to the distributed matrix *A*.

tau	(local)
	`REAL` for `psgeqpf`
	`DOUBLE PRECISION` for `pdgeqpf`
	`COMPLEX` for `pcgeqpf`
	`DOUBLE COMPLEX` for `pzgeqpf`.
	Array of size *LOCc*(*ja*+min(*m*, *n*)-1).
	Contains the scalar factor *tau* of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
rwork(1)	On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance.
info	(global) `INTEGER`. = 0, the execution is successful. < 0, if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1)**H*(2)*...**H*(*k*)

where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H = *I* - *tau***v***v'*

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+*1*:*m*) is stored on exit in *A*(*ia*+*i*:*ia*+m-1, *ja*+*i*-1).

The matrix *P* is represented in *ipiv* as follows: if *ipiv*(*j*)= *i* then the *j*-th column of *P* is the *i*-th canonical unit vector.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orgqr

Generates the orthogonal matrix Q of the QR factorization formed by `p?geqrf`.

Syntax

```
call psorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The p?orgqrroutine generates the whole or part of *m*-by-*n* real distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal columns, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*

Q= *H*(1)**H*(2)*...**H*(*k*)

as returned by p?geqrf.

Input Parameters

m — (global) INTEGER. The number of rows in the matrix sub(*Q*) (*m*≥ 0).

n — (global) INTEGER. The number of columns in the matrix sub(*Q*) (*m*≥*n*≥ 0).

k — (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*n*≥*k*≥ 0).

a — (local)

REAL for psorgqr

DOUBLE PRECISION for pdorgqr

Pointer into the local memory to an array of local size (*lld_a*, *LOCc*(*ja*+*n*-1)). The *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja*≤ *j* ≤ *ja* +*k*-1, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*:*ja*+*k*-1).

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau — (local)

REAL for psorgqr

DOUBLE PRECISION for pdorgqr

Array of size *LOCc*(*ja*+*k*-1).

Contains the scalar factor *tau*(*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.

work — (local)

REAL for psorgqr

DOUBLE PRECISION for pdorgqr

Workspace array of size of *lwork*.

lwork — (local or global) INTEGER, size of *work*.

Must be at least *lwork*≥*nb_a**(*nqa*0 + *mpa*0 + *nb_a*), where

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL);

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — Contains the local pieces of the *m*-by-*n* distributed matrix Q.

work(1) — On exit, (1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call pcungqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

This routine generates the whole or part of *m*-by-*n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal columns, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*

Q = *H*(1)**H*(2)*...**H*(*k*)

as returned by p?geqrf.

Input Parameters

m — (global) INTEGER. The number of rows in the matrix sub(*Q*); (*m*≥0).

n — (global) INTEGER. The number of columns in the matrix sub(*Q*) (*m*≥*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*n*≥*k*≥0).

a (local)

COMPLEX for pcungqr

DOUBLE COMPLEX for pzungqr

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). The *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja*≤ *j*≤ *ja* +*k*-1, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*:*ja*+*k*-1).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcungqr

DOUBLE COMPLEX for pzungqr

Array of size *LOCc*(*ja*+*k*-1).

Contains the scalar factor *tau*(*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.

work (local)

COMPLEX for pcungqr

DOUBLE COMPLEX for pzungqr

Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least *lwork*≥*nb_a**(*nqa*0 + *mpa*0 + *nb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	Contains the local pieces of the *m*-by-*n* distributed matrix *Q*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormqr

Multiplies a general matrix by the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

call psormqr(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdormqr(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Include Files

Description

The p?ormqrroutine overwrites the general real *m*-by-*n* distributed matrix sub (*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^T*sub(*C*)	sub(*C*)*Q^T

where *Q* is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1) *H*(2)... *H*(*k*)

as returned by p?geqrf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	(global) CHARACTER ='L': *Q* or Q^T is applied from the left. ='R': *Q* or Q^T is applied from the right.
trans	(global) CHARACTER ='N', no transpose, *Q* is applied. ='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m*≥*k*≥0

If *side* = 'R', *n*≥*k*≥0.

a (local)

REAL for psormqr

DOUBLE PRECISION for pdormqr.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). The *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja*≤*j*≤*ja*+*k*-1, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*:*ja*+*k*-1). *A*(*ia*:*, *ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

If *side* = 'L', *lld_a* ≥ max(1, *LOCr*(*ia*+*m*-1))

If *side* = 'R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*-1))

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

REAL for psormqr

DOUBLE PRECISION for pdormqr

Array of size *LOCc*(*ja*+*k*-1).

Contains the scalar factor *tau*(*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.

c (local)

REAL for psormqr

DOUBLE PRECISION for pdormqr

Pointer into the local memory to an array of local size (*lld_c*, *LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the matrix sub(*C*), respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

REAL for psormqr

DOUBLE PRECISION for pdormqr.

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

if *side* = 'L',

lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0+*mpc*0)**nb_a*) + *nb_a***nb_a*

else if *side* = 'R',

lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0+max(*npa*0+numroc(numroc(*n*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if

where

lcmq = *lcm*/NPCOL with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0= numroc(*n*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = mod(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0= numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0= numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c

Overwritten by the product Q*sub(C), or Q^T*sub(C), or sub(C)*Q^T, or sub(C)*Q.

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by p?geqrf.

Syntax

call pcunmqr(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmqr(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Include Files

Description

This routine overwrites the general complex *m*-by-*n* distributed matrix sub (*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1) *H*(2)... *H*(*k*) as returned by p?geqrf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* ='R'.

Input Parameters

side (global) CHARACTER

='L': *Q* or Q^H is applied from the left.

='R': *Q* or Q^H is applied from the right.

trans (global) CHARACTER

='N', no transpose, *Q* is applied.

='C', conjugate transpose, Q^H is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m*≥*k*≥0

If *side* = 'R', *n*≥*k*≥0.

a (local)

COMPLEX for pcunmqr

DOUBLE COMPLEX for pzunmqr.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*k*-1)). The *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja*≤*j*≤*ja*+*k*-1, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*:*ja*+*k*-1). *A*(*ia*:*, *ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

If *side* = 'L', *lld_a* ≥ max(1, *LOCr*(*ia*+*m*-1))

If *side* = 'R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*-1))

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcunmqr

DOUBLE COMPLEX for pzunmqr

Array of size *LOCc*(*ja*+*k*-1).

Contains the scalar factor *tau*(*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.

c (local)

COMPLEX for pcunmqr

DOUBLE COMPLEX for pzunmqr.

Pointer into the local memory to an array of local size (*lld_c*,*LOCc*(*jc* +*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for pcunmqr

DOUBLE COMPLEX for pzunmqr.

Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least:

If *side* = 'L',

lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a***nb_a*

else if *side* = 'R',

lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*n*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if

where

lcmq = *lcm*/NPCOL with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = numroc(*n*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c

Overwritten by the product *Q**sub(*C*), or Q^H*sub(*C*), or sub(*C*)*Q^H, or sub(*C*)**Q* .

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gelqf

Computes the LQ factorization of a general rectangular matrix.

Syntax

```
call psgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The p?gelqf routine computes the *LQ* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*)= *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) = *L***Q*.

Input Parameters

m

(global) INTEGER. The number of rows in the distributed submatrix sub(*A*) (*m*≥ 0).

n

(global) INTEGER. The number of columns in the distributed submatrix sub(*A*) (*n*≥ 0).

a

(local)

REAL for psgelqf

DOUBLE PRECISION for pdgelqf

COMPLEX for pcgelqf

DOUBLE COMPLEX for pzgelqf

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja*

(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work

(local)

REAL for psgelqf

DOUBLE PRECISION for pdgelqf

COMPLEX for pcgelqf

DOUBLE COMPLEX for pzgelqf

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least *lwork*≥*mb_a**(*mp*0 + *nq*0 + *mb_a*), where

iroff = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

The elements on and below the diagonal of sub(*A*) contain the *m*-by-min(*m*,*n*) lower trapezoidal matrix *L* (*L* is lower trapezoidal if *m* ≤ *n*); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau

(local)

REAL for psgelqf

DOUBLE PRECISION for pdgelqf

COMPLEX for pcgelqf

DOUBLE COMPLEX for pzgelqf

Array of size *LOCr*(*ia*+min(*m*, *n*)-1).

Contains the scalar factors of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ia*+*k*-1)**H*(*ia*+*k*-2)*...**H*(*ia*),

where $k = \min(m, n)$

Each $H(i)$ has the form

$H(i) = I - tau*v*v'$

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$, and *tau* in $tau(ia+i-1)$.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by p?gelqf*.*

Syntax

```
call psorglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The p?orglq routine generates the whole or part of m-by-n real distributed matrix *Q* denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = H(k)*...* H(2)* H(1)$

as returned by p?gelqf.

Input Parameters

m	(global) INTEGER. The number of rows in the matrix sub(*Q*); $(m \geq 0)$.
n	(global) INTEGER. The number of columns in the matrix sub(*Q*) $(n \geq m \geq 0)$.
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* $(m \geq k \geq 0)$.
a	(local) REAL for psorglq DOUBLE PRECISION for pdorglq Pointer into the local memory to an array of local size $(lld\_a, LOCc(ja+n-1))$. On entry, the *i*-th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.
ia, ja	(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
$desca$	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work (local)

REAL for psorglq

DOUBLE PRECISION for pdorglq

Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least *lwork*≥*mb_a**(*mpa*0+*nqa*0+*mb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.

tau (local)

REAL for psorglq

DOUBLE PRECISION for pdorglq

Array of size *LOCr*(*ia*+*k*-1).

Contains the scalar factors *tau*(*j*) of elementary reflectors *H*(*j*). *tau* is tied to the distributed matrix *A*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unglq

Generates the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
call pcunglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzunglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

This routine generates the whole or part of m-by-n complex distributed matrix Q denoting $A(ia{:}ia+m-1, ja{:}ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = (H(k))^H ... *(H(2))^H*(H(1))^H$ as returned by p?gelqf.

Input Parameters

m	(global) INTEGER. The number of rows in the matrix sub(Q) ($m \geq 0$).
n	(global) INTEGER. The number of columns in the matrix sub(Q) ($n \geq m \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
a	(local) COMPLEX for pcunglq DOUBLE COMPLEX for pzunglq Pointer into the local memory to an array of local size (lld_a, $LOCc(ja+n-1)$). On entry, the i-th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its distributed matrix argument $A(ia{:}ia+k-1, ja{:}*)$.
ia, *ja*	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia{:}ia+m-1, ja{:}ja+n-1)$, respectively.
desca	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.
tau	(local) COMPLEX for pcunglq DOUBLE COMPLEX for pzunglq Array of size $LOCr(ia+k-1)$. Contains the scalar factors $tau(j)$ of elementary reflectors $H(j)$. tau is tied to the distributed matrix A.
work	(local) COMPLEX for pcunglq DOUBLE COMPLEX for pzunglq

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least *lwork*≥*mb_a**(*mpa*0+*nqa*0+*mb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormlq

Multiplies a general matrix by the orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

call psormlq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *work*, *lwork*, *info*)

call pdormlq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *work*, *lwork*, *info*)

Include Files

Description

The p?ormlq routine overwrites the general real *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^T*sub(*C*)	sub(*C*)*Q^T

where *Q* is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(*k*)...*H*(2) *H*(1)

as returned by p?gelqf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER

='L': *Q* or Q^T is applied from the left.

='R': *Q* or Q^T is applied from the right.

trans (global) CHARACTER

='N', no transpose, *Q* is applied.

='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m*≥*k*≥0

If *side* = 'R', *n*≥*k*≥0.

a (local)

REAL for psormlq

DOUBLE PRECISION for pdormlq.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)), if *side* = 'L' and (*lld_a*,*LOCc*(*ja*+*n*-1)), if *side* = 'R'. The *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*-1, as returned by p?gelqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1, *ja*:*).

A(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

REAL for psormlq

DOUBLE PRECISION for pdormlq

Array of size *LOCc*(*ja*+*k*-1).

Contains the scalar factor *tau*(*j*) of elementary reflectors *H*(*j*) as returned by p?gelqf. *tau* is tied to the distributed matrix *A*.

c (local)

REAL for psormlq

DOUBLE PRECISION for pdormlq

Pointer into the local memory to an array of local size (*lld_c*,*LOCc*(*jc* +*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)

REAL for psormlq

DOUBLE PRECISION for pdormlq.

Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of the array *work*; must be at least:

If *side* = 'L',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0+max*mqa*0)+ numroc(numroc(*m* + *iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))* *mb_a*) + *mb_a*mb_a*

else if *side* = 'R',

lwork≥max((*mb_a** (*mb_a*-1))/2, (*mpc*0+nqc0)*mb_a + *mb_a*mb_a*

end if

where

lcmp = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = *numroc*(*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q**sub(*C*), or *Q*' *sub (*C*), or sub(*C*)**Q*', or sub(*C*)**Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmlq

Multiplies a general matrix by the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax

call pcunmlq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmlq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Include Files

Description

This routine overwrites the general complex *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

side ='L' *side* ='R'

trans = 'N':	Q*sub(C)	sub(C)*Q
trans = 'T':	Q^H*sub(C)	sub(C)*Q^H

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k)' \ldots H(2)' H(1)'$

as returned by p?gelqf. Q is of order m if $side$ = 'L' and of order n if $side$ = 'R'.

Input Parameters

side
(global) CHARACTER
='L': Q or Q^H is applied from the left.
='R': Q or Q^H is applied from the right.

trans
(global) CHARACTER
='N', no transpose, Q is applied.
='C', conjugate transpose, Q^H is applied.

m
(global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n
(global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k
(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:
If $side$ = 'L', $m \geq k \geq 0$
If $side$ = 'R', $n \geq k \geq 0$.

a
(local)
COMPLEX for pcunmlq
DOUBLE COMPLEX for pzunmlq.
Pointer into the local memory to an array of size (lld_a, $LOCc(ja+m-1)$), if $side$ = 'L' and (lld_a, $LOCc(ja+n-1)$), if $side$ = 'R', where $lld\_a \geq \max(1, LOCr(ia+k-1))$. The i-th column must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.

ia, *ja*
(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.

desca
(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

tau
(local)
COMPLEX for pcunmlq
DOUBLE COMPLEX for pzunmlq
Array of size $LOCc(ia+k-1)$.

Contains the scalar factor *tau*(*j*) of elementary reflectors *H*(*j*) as returned by p?gelqf. *tau* is tied to the distributed matrix *A*.

c

(local)

COMPLEX for pcunmlq

DOUBLE COMPLEX for pzunmlq.

Pointer into the local memory to an array of local size (*lld_c*,*LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc*

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

COMPLEX for pcunmlq

DOUBLE COMPLEX for pzunmlq.

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of the array *work*; must be at least:

If *side* = 'L',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max*mqa*0)+ numroc(numroc(*m* + *iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* = 'R',

lwork≥max((*mb_a** (*mb_a*-1))/2, (*mpc*0 + nqc0)**mb_a* + *mb_a***mb_a*

end if

where

lcmp = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = *numroc*(m + *icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
`mod(x,y)` is the integer remainder of *x*/*y*.

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c — Overwritten by the product *Q**sub(*C*), or *Q'**sub (*C*), or sub(*C*)**Q'*, or sub(*C*)**Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) `INTEGER`.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?geqlf

Computes the QL factorization of a general matrix.

Syntax

```
call psgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The `p?geqlf` routine forms the *QL* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*)= *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) = *Q***L*.

Input Parameters

m — (global) `INTEGER`. The number of rows in the matrix sub(*Q*); (*m*≥ 0).

n — (global) `INTEGER`. The number of columns in the matrix sub(*Q*) (*n*≥ 0).

a (local)

REAL for psgeqlf

DOUBLE PRECISION for pdgeqlf

COMPLEX for pcgeqlf

DOUBLE COMPLEX for pzgeqlf

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja*+*n*-1)). Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work (local)

REAL for psgeqlf

DOUBLE PRECISION for pdgeqlf

COMPLEX for pcgeqlf

DOUBLE COMPLEX for pzgeqlf

Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least *lwork*≥*nb_a**(*mp*0 + *nq*0 + *nb_a*), where

iroff = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

numroc and indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	On exit, if *m*≥*n*, the lower triangle of the distributed submatrix *A*(*ia*+*m*-*n*:*ia*+*m*-1, *ja*:*ja*+*n*-1) contains the *n*-by-*n* lower triangular matrix *L*; if *m*≤*n*, the elements on and below the (*n* - *m*)-th superdiagonal contain the *m*-by-*n* lower trapezoidal matrix *L*; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau	(local)
	`REAL` for `psgeqlf`
	`DOUBLE PRECISION` for `pdgeqlf`
	`COMPLEX` for `pcgeqlf`
	`DOUBLE COMPLEX` for `pzgeqlf`
	Array of size *LOCc*(*ja*+*n*-1).
	Contains the scalar factors of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) `INTEGER`. `= 0`: the execution is successful. `< 0`: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ja*+*k*-1)*...**H*(*ja*+1)**H*(*ja*)

where *k* = `min`(*m*,*n*)

Each *H*(*i*) has the form

H(i) = *I* - *tau***v***v'*

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*m*-*k*+*i*+1:*m*) = 0 and *v*(*m*-*k*+*i*) = 1; *v*(1:*m*-*k*+*i*-1) is stored on exit in *A*(*ia*:*ia*+*m*-*k*+*i*-2, *ja*+*n*-*k*+*i*-1), and *tau* in *tau*(*ja*+*n*-*k*+*i*-1).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orgql

Generates the orthogonal matrix Q of the QL factorization formed by `p?geqlf`.

Syntax

```
call psorgql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The p?orgql routine generates the whole or part of *m*-by-*n* real distributed matrix *Q* denoting *A*(*ia*:*ia* +*m*-1,*ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the first *m* rows of a product of *k* elementary reflectors of order *n*

Q = *H*(*k*)*...*H*(2)*H*(1)

as returned by p?geqlf.

Input Parameters

m	(global) INTEGER. The number of rows in the matrix sub(*Q*), (*m*≥0).
n	(global) INTEGER. The number of columns in the matrix sub(*Q*), (*m*≥*n*≥0).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*n*≥*k*≥0).
a	(local) REAL for psorgql DOUBLE PRECISION for pdorgql Pointer into the local memory to an array of local size (*lld_a*, *LOCc*(*ja* +*n*-1)). On entry, the *j*-th column must contain the vector that defines the elementary reflector *H*(*j*),*ja*+*n*-*k*≤*j*≤*ja*+*n*-1, as returned by p?geqlf in the *k* columns of its distributed matrix argument *A*(*ia*:*,*ja*+*n*-*k*:*ja*+*n*-1).
ia, *ja*	(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia* +*m*-1,*ja*:*ja*+*n*-1), respectively.
desca	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.
tau	(local) REAL for psorgql DOUBLE PRECISION for pdorgql Array of size *LOCc*(*ja*+*n*-1). Contains the scalar factors *tau*(*j*) of elementary reflectors *H*(*j*). *tau* is tied to the distributed matrix *A*.
work	(local) REAL for psorgql DOUBLE PRECISION for pdorgql Workspace array of size of *lwork*.
lwork	(local or global) INTEGER, size of *work*, must be at least *lwork*≥*nb_a*(*nqa*0+*mpa*0+*nb_a*), where *iroffa* = mod(*ia*-1, *mb_a*),

```
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL)
```

NOTE
`mod(x,y)` is the integer remainder of x/y.

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a
Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.

work(1)
On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info
(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ungql

Generates the unitary matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call pcungql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

This routine generates the whole or part of *m*-by-*n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*

$Q = (H(k))^H ... *(H(2))^H*(H(1))^H$ as returned by p?geqlf.

Input Parameters

m (global) INTEGER. The number of rows in the matrix sub(*Q*) (*m*≥0).

n (global) INTEGER. The number of columns in the matrix sub(*Q*) (*m*≥*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*n*≥*k*≥0).

a (local)

COMPLEX for pcungql

DOUBLE COMPLEX for pzungql

Pointer into the local memory to an array of local size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja*+*n*-*k*≤ *j*≤ *ja*+*n*-1, as returned by p?geqlf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*+*n*-*k*: *ja*+*n*-1).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcungql

DOUBLE COMPLEX for pzungql

Array of size *LOCr*(*ia*+*n*-1).

Contains the scalar factors *tau*(*j*) of elementary reflectors *H*(*j*). *tau* is tied to the distributed matrix *A*.

work (local)

COMPLEX for pcungql

DOUBLE COMPLEX for pzungql

Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least *lwork*≥*nb_a**(*nqa*0 + *mpa*0 + *nb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormql

Multiplies a general matrix by the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call psormql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
call pdormql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

Include Files

Description

The p?ormqlroutine overwrites the general real *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^T*sub(*C*)	sub(*C*)*Q^T

where *Q* is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(*k*)' ... *H*(2)' *H*(1)'

as returned by p?geqlf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — (global) CHARACTER

='L': *Q* or Q^T is applied from the left.

='R': Q or Q^T is applied from the right.

trans (global) CHARACTER

='N', no transpose, Q is applied.

='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C), ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(C), ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:

If *side* = 'L', $m \geq k \geq 0$

If *side* = 'R', $n \geq k \geq 0$.

a (local)

REAL for psormql

DOUBLE PRECISION for pdormql.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*k*-1)). The j-th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?gelqf in the k columns of its distributed matrix argument $A(ia{:}*, ja{:}ja+k-1)$. $A(ia{:}*, ja{:}ja+k-1)$ is modified by the routine but restored on exit.

If *side* = 'L',*lld_a* ≥ max(1, *LOCr*(*ia*+*m*-1)),

If *side* = 'R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

tau (local)

REAL for psormql

DOUBLE PRECISION for pdormql.

Array of size *LOCc*(*ja*+*n*-1).

Contains the scalar factor *tau*(j) of elementary reflectors $H(j)$ as returned by p?geqlf. *tau* is tied to the distributed matrix A.

c (local)

REAL for psormql

DOUBLE PRECISION for pdormql.

Pointer into the local memory to an array of local size (*lld_c*,*LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(C) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)

REAL for psormql

DOUBLE PRECISION for pdormql.

Workspace array of size of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

If *side* = 'L',

lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0+mpc0)**nb_a* + *nb_a***nb_a*

else if *side* ='R',

lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0+max(*npa*0 + numroc(numroc(*n*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if

where

lcmq = *lcm*/NPCOL with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0= *numroc*(*n* + *iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q** sub(*C*), or *Q*'*sub (*C*), or sub(*C*)* *Q*', or sub(*C*)* *Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmql

Multiplies a general matrix by the unitary matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call pcunmql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

```
call pzunmql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

Include Files

Description

This routine overwrites the general complex *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(*k*)' ... *H*(2)' *H*(1)'

as returned by p?geqlf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	(global) CHARACTER ='L': *Q* or Q^H is applied from the left. ='R': *Q* or Q^H is applied from the right.
trans	(global) CHARACTER ='N', no transpose, *Q* is applied.

='C', conjugate transpose, Q^H is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m*≥*k*≥0

If *side* = 'R', *n*≥*k*≥0.

a (local)

COMPLEX for pcunmql

DOUBLE COMPLEX for pzunmql.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*k*-1)). The *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja*≤*j*≤*ja*+*k*-1, as returned by p?geqlf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*:*ja*+*k*-1). *A*(*ia*:*, *ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

If *side* = 'L',*lld_a* ≥ max(1, *LOCr*(*ia*+*m*-1)),

If *side* = 'R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcunmql

DOUBLE COMPLEX for pzunmql

Array of size *LOCc*(*ia*+*n*-1).

Contains the scalar factor *tau*(*j*) of elementary reflectors *H*(*j*) as returned by p?geqlf. *tau* is tied to the distributed matrix *A*.

c (local)

COMPLEX for pcunmql

DOUBLE COMPLEX for pzunmql.

Pointer into the local memory to an array of local size (*lld_c*,*LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for pcunmql

DOUBLE COMPLEX for pzunmql.

Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least:

If *side* = 'L',

lwork≥max((*nb_a** (*nb_a*-1))/2, (*nqc*0+mpc0)**nb_a* + *nb_a***nb_a*

else if *side* ='R',

lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0+max*npa*0)+ numroc(numroc(*n* +*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if

where

lcmp = *lcm*/NPCOL with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = *numroc* (*n* + *iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q** sub(*C*), or *Q*' sub (*C*), or sub(*C*)* *Q*', or sub(*C*)* *Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gerqf

Computes the RQ factorization of a general rectangular matrix.

Syntax

```
call psgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The p?gerqf routine forms the *QR* factorization of a general *m*-by-*n* distributed matrix sub(*A*)= *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) as

A= *R***Q*

Input Parameters

m — (global) INTEGER. The number of rows in the distributed matrix sub(*A*); (*m*≥0).

n — (global) INTEGER. The number of columns in the distributed matrix sub(*A*); (*n*≥0).

a — (local)

REAL for psgeqrf

DOUBLE PRECISION for pdgeqrf

COMPLEX for pcgeqrf

DOUBLE COMPLEX for pzgeqrf.

Pointer into the local memory to an array of local size (*lld_a*,*LOCc*(*ja*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*

work (local).

REAL for psgeqrf

DOUBLE PRECISION for pdgeqrf.

COMPLEX for pcgeqrf.

DOUBLE COMPLEX for pzgeqrf

Workspace array of size *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least *lwork*≥*mb_a**(*mp*0+*nq*0+*mb_a*), where

iroff = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL) and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, if *m*≤*n*, the upper triangle of *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) contains the *m*-by-*m* upper triangular matrix *R*; if *m*≥*n*, the elements on and above the (*m* - *n*)-th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*; the

	remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).
tau	(local) REAL for psgeqrf DOUBLE PRECISION for pdgeqrf COMPLEX for pcgeqrf DOUBLE COMPLEX for pzgeqrf. Array of size *LOCr*(*ia*+*m*-1). Contains the scalar factor of elementary reflectors. *tau* is tied to the distributed matrix *A*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0, the execution is successful. < 0, if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ia*)**H*(*ia*+1)*...**H*(*ia*+*k*-1),

where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau***v***v'*

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*n*-*k*+*i*+1:*n*) = 0 and *v*(*n*-*k*+*i*) = 1; *v*(1:*n*-*k*+*i*-1) is stored on exit in *A*(*ia*+*m*-*k*+*i*-1,*ja*:*ja*+*n*-*k*+*i*-2), and *tau* in *tau*(*ia*+*m*-*k*+*i*-1).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orgrq

Generates the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call psorgrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The p?orgrqroutine generates the whole or part of *m*-by-*n* real distributed matrix *Q* denoting *A*(*ia*:*ia* +*m*-1,*ja*:*ja*+*n*-1) with orthonormal rows that is defined as the last *m* rows of a product of *k* elementary reflectors of order *n*

Q= *H*(1)**H*(2)*...**H*(*k*)

as returned by p?gerqf.

Input Parameters

m — (global) INTEGER. The number of rows in the matrix sub(*Q*), (*m*≥0).

n — (global) INTEGER. The number of columns in the matrix sub(*Q*), (*n*≥*m*≥0).

k — (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*m*≥*k*≥0).

a — (local)

REAL for psorgrq

DOUBLE PRECISION for pdorgrq

Pointer into the local memory to an array of local size (*lld_a*, *LOCc*(*ja* +*n*-1)). The *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia*≤i≤*ia*+*m*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*+*m*-*k*:*ia*+*m*-1, *ja*:*).

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau — (local)

REAL for psorgrq

DOUBLE PRECISION for pdorgrq

Array of size *LOCc*(*ja*+*k*-1).

Contains the scalar factor *tau*(*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

work — (local)

REAL for psorgrq

DOUBLE PRECISION for pdorgrq

Workspace array of size of *lwork*.

lwork — (local or global) INTEGER, size of *work*, must be at least *lwork*≥*mb_a**(*mpa*0 + *nqa*0 + *mb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — Contains the local pieces of the *m*-by-*n* distributed matrix Q.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ungrq

Generates the unitary matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call pcungrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

This routine generates the *m*-by-*n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *n*

$Q = (H(1))^H*(H(2))^H*...*(H(k))^H$ as returned by p?gerqf.

Input Parameters

m — (global) INTEGER. The number of rows in the matrix sub(*Q*); (*m*≥0).

n (global) INTEGER. The number of columns in the matrix sub(*Q*) (*n*≥*m*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*m*≥*k*≥0).

a (local)

COMPLEX for pcungrq

DOUBLE COMPLEX for pzungrqc

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). The *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia*+*m*-*k*≤*i*≤*ia*+*m*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*+*m*-*k*:*ia*+*m*-1, *ja*:*).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcungrq

DOUBLE COMPLEX for pzungrq

Array of size *LOCr*(*ia*+*m*-1).

Contains the scalar factor *tau*(*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

work (local)

COMPLEX for pcungrq

DOUBLE COMPLEX for pzungrq

Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least *lwork*≥*mb_a**(*mpa*0 +*nqa*0+*mb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — Contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormr3

Applies an orthogonal distributed matrix to a general m-by-n distributed matrix.

Syntax

```
call psormr3 (side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info )
```

```
call pdormr3 (side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info )
```

Description

p?ormr3 overwrites the general real *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N'	*Q* * sub(*C*)	*sub(C)* * *Q*
trans = 'T'	Q^T * sub(*C*) *Q* * sub(*C*)	sub(*C*) * Q^T

where *Q* is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

Q = H(1) H(2) . . . H(*k*)

as returned by p?tzrzf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — (global)

CHARACTER.

= 'L': apply *Q* or Q^T from the Left;

= 'R': apply Q or Q^T from the Right.

trans (global)

CHARACTER.

= 'N': No transpose, apply Q;

= 'T': Transpose, apply Q^T.

m (global)

INTEGER.

The number of rows to be operated on i.e the number of rows of the distributed submatrix sub(*C*). *m* >= 0.

n (global)

INTEGER.

The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(*C*). *n* >= 0.

k (global)

INTEGER.

The number of elementary reflectors whose product defines the matrix Q.

If *side* = 'L', *m* >= *k* >= 0,

if *side* = 'R', *n* >= *k* >= 0.

l (global)

INTEGER.

The columns of the distributed submatrix sub(*A*) containing the meaningful part of the Householder reflectors.

If *side* = 'L', *m* >= *l* >= 0,

if *side* = 'R', *n* >= *l* >= 0.

a (local)

`REAL` for `psormr3`

`DOUBLE PRECISION` for `pdormr3`

Pointer into the local memory to an array of size `(lld_a,LOCc(ja+m-1))` if *side*='L', and `(lld_a,LOCc(ja+n-1))` if *side*='R', where *lld_a* >= MAX(1,LOCr(*ia*+*k*-1));

On entry, the *i*-th row must contain the vector which defines the elementary reflector H(*i*), *ia* <= *i* <= *ia*+*k*-1, as returned by `p?tzrzf` in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1,*ja*:*).

A(*ia*:*ia*+*k*-1,*ja*:*) is modified by the routine but restored on exit.

ia (global)

INTEGER.

The row index in the global array *a* indicating the first row of sub(*A*).

ja (global)

INTEGER.

The column index in the global array *a* indicating the first column of sub(*A*).

desca (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

tau (local)

REAL for psormr3

DOUBLE PRECISION for pdormr3

Array, size LOCc(*ia*+*k*-1).

This array contains the scalar factors *tau*(i) of the elementary reflectors H(i) as returned by p?tzrzf. *tau* is tied to the distributed matrix *A*.

c (local)

REAL for psormr3

DOUBLE PRECISION for pdormr3

Pointer into the local memory to an array of size (*lld_c*,*LOCc*(*jc*+*n*-1)) .

On entry, the local pieces of the distributed matrix sub(*C*).

ic (global)

INTEGER.

The row index in the global array *c* indicating the first row of sub(*C*).

jc (global)

INTEGER.

The column index in the global array *c* indicating the first column of sub(*C*).

descc (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *C*.

work (local)

REAL for psormr3

DOUBLE PRECISION for pdormr3

Array, size (*lwork*)

lwork (local)

INTEGER.

The size of the array *work*.

lwork is local input and must be at least

If *side* = 'L', *lwork* >= *MpC0* + MAX(MAX(1, *NqC0*), numroc(numroc(*m* +*IROFFC*,*mb_a*,0,0,*NPROW*),*mb_a*,0,0,*NqC0*));

if *side* = 'R', *lwork* >= *NqC0* + MAX(1, *MpC0*);

where *LCMP* = *LCM* / *NPROW*

LCM = iclm(*NPROW*, *NPCOL*),

IROFFC = MOD(*ic*-1, *mb_c*),

ICOFFC = MOD(*jc*-1, *nb_c*),

ICROW = indxg2p(*ic*, *mb_c*, *MYROW*, *rsrc_c*, *NPROW*),

ICCOL = indxg2p(*jc*, *nb_c*, *MYCOL*, *csrc_c*, *NPCOL*),

MpC0 = numroc(*m*+*IROFFC*, *mb_c*, *MYROW*, *ICROW*, *NPROW*),

NqC0 = numroc(*n*+*ICOFFC*, *nb_c*, *MYCOL*, *ICCOL*, *NPCOL*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions;

MYROW, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c

On exit, sub(*C*) is overwritten by *Q**sub(*C*) or *Q*'*sub(*C*) or sub(*C*)**Q*' or sub(*C*)**Q*.

work

On exit, *work*(1) returns the minimal and optimal *lwork*.

info

(local)

INTEGER.

= 0: successful exit

< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

Alignment requirements

The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L',

(*nb_a* = *mb_c* .AND. ICOFFA = *IROFFC*)

If *side* = 'R',

(*nb_a* = *nb_c* .AND. ICOFFA = *ICOFFC* .AND. IACOL = *ICCOL*)

p?unmr3

Applies an orthogonal distributed matrix to a general m-by-n distributed matrix.

Syntax

```
call pcunmr3 (side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info )
```

```
call pzunmr3 (side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info )
```

Description

p?unmr3 overwrites the general complex *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	*Q* * sub(*C*)	sub(*C*) * *Q*
trans = 'C':	Q^H * sub(*C*)	sub(*C*) * Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = H(1)' H(2)' . . . H(*k*)'

as returned by p?tzrzf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global)

CHARACTER.

= 'L': apply *Q* or Q^H from the Left;

= 'R': apply *Q* or Q^H from the Right.

trans (global)

CHARACTER.

= 'N': No transpose, apply *Q*;

= 'C': Conjugate transpose, apply Q^H.

m (global)

INTEGER.

The number of rows to be operated on i.e the number of rows of the distributed submatrix sub(*C*). *m* >= 0.

n (global)

INTEGER.

The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(*C*). *n* >= 0.

k (global)

INTEGER.

The number of elementary reflectors whose product defines the matrix *Q*.

If *side* = 'L', *m* >= *k* >= 0, if *side* = 'R', *n* >= *k* >= 0.

l (global)

INTEGER.

The columns of the distributed submatrix *sub(A)* containing the meaningful part of the Householder reflectors.

If *side* = 'L', *m* >= *l* >= 0, if *side* = 'R', *n* >= *l* >= 0.

a (local)

COMPLEX for pcunmr3

DOUBLE COMPLEX for pzunmr3

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)) if *side*='L', and (*lld_a*,*LOCc*(*ja*+*n*-1)) if *side*='R', where *lld_a* >= MAX(1,LOCr(*ia*+*k*-1));

On entry, the i-th row must contain the vector which defines the elementary reflector H(i), *ia* <= i <= *ia*+*k*-1, as returned by p?tzrzf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1,*ja*:*).

A(*ia*:*ia*+*k*-1,*ja*:*) is modified by the routine but restored on exit.

ia (global)

INTEGER.

The row index in the global array *a* indicating the first row of *sub(A)*.

ja (global)

INTEGER.

The column index in the global array *a* indicating the first column of *sub(A)*.

desca (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcunmr3

DOUBLE COMPLEX for pzunmr3

Array, size LOCc(*ia*+*k*-1).

This array contains the scalar factors *tau*(i) of the elementary reflectors H(i) as returned by p?tzrzf. *tau* is tied to the distributed matrix *A*.

c (local)

COMPLEX for pcunmr3

DOUBLE COMPLEX for pzunmr3

Pointer into the local memory to an array of size (*lld_c*,*LOCc*(*jc*+*n*-1)) .

On entry, the local pieces of the distributed matrix sub(*C*).

ic (global)

INTEGER.

The row index in the global array *c* indicating the first row of sub(*C*).

jc (global)

INTEGER.

The column index in the global array *c* indicating the first column of sub(*C*).

descc (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for pcunmr3

DOUBLE COMPLEX for pzunmr3

Array, size (*lwork*)

On exit, *work*(1) returns the minimal and optimal *lwork*.

lwork (local or global)

INTEGER.

The size of the array *work*.

lwork is local input and must be at least

If *side* = 'L', *lwork* >= MpC0 + MAX(MAX(1, NqC0), numroc(numroc(*m* +IROFFC,*mb_a*,0,0,NPROW),*mb_a*,0,0,LCMP));

if *side* = 'R', *lwork* >= NqC0 + MAX(1, MpC0);

where LCMP = LCM / NPROW with LCM = ICLM(NPROW, NPCOL),

IROFFC = MOD(*ic*-1, MB_C), ICOFFC = MOD(*jc*-1, *nb_c*),

ICROW = indxg2p(*ic*, MB_C, MYROW, *rsrc_c*, NPROW),

ICCOL = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

MpC0 = numroc(*m*+IROFFC, MB_C, MYROW, ICROW, NPROW),

NqC0 = numroc(*n*+ICOFFC, *nb_c*, MYCOL, ICCOL, NPCOL),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions;

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	On exit, sub(*C*) is overwritten by *Q**sub(*C*) or *Q*'*sub(*C*) or sub(*C*)**Q*' or sub(*C*)**Q*.
work	(local) COMPLEX for pcunmr3 DOUBLE COMPLEX for pzunmr3 Array, size (*lwork*) On exit, *work*(1) returns the minimal and optimal *lwork*.
info	(local) INTEGER. = 0: successful exit < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

Alignment requirements

The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*nb_a* = MB_C and ICOFFA = IROFFC)

If *side* = 'R', (*nb_a* = *nb_c* and ICOFFA = ICOFFC and IACOL = ICCOL)

p?ormrq

Multiplies a general matrix by the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

call psormrq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdormrq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Include Files

Description

The p?ormrqroutine overwrites the general real *m*-by-*n* distributed matrix sub (*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^T*sub(*C*)	sub(*C*)*Q^T

where *Q* is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1) *H*(2)... *H*(*k*)

as returned by p?gerqf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER

='L': *Q* or Q^T is applied from the left.

='R': *Q* or Q^T is applied from the right.

trans (global) CHARACTER

='N', no transpose, *Q* is applied.

='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m*≥*k*≥0

If *side* = 'R', *n*≥*k*≥0.

a (local)

REAL for psormqr

DOUBLE PRECISION for pdormqr.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*,*LOCc*(*ja*+*n*-1)) if *side* = 'R'.

The *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1, *ja*:*). *A*(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

REAL for psormqr

DOUBLE PRECISION for pdormqr

Array of size *LOCc*(*ja*+*k*-1).

Contains the scalar factor *tau*(*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

c (local)

REAL for psormrq

DOUBLE PRECISION for pdormrq

Pointer into the local memory to an array of local size (*lld_c*,*LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc*

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the matrix sub(*C*), respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

REAL for psormrq

DOUBLE PRECISION for pdormrq.

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

If *side* = 'L',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0 + numroc(numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* ='R',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a***mb_a*

end if

where

lcmp = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q** sub(*C*), or *Q*'*sub (*C*), or sub(*C*)* *Q*', or sub(*C*)* *Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmrq

Multiplies a general matrix by the unitary matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call pcunmrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

```
call pzunmrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

Include Files

Description

This routine overwrites the general complex *m*-by-*n* distributed matrix sub (*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1)' *H*(2)'... *H*(*k*)'

as returned by p?gerqf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — (global) CHARACTER

='L': *Q* or Q^H is applied from the left.

='R': Q or Q^H is applied from the right.

trans (global) CHARACTER

='N', no transpose, Q is applied.

='C', conjugate transpose, Q^H is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C) , ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(C), ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:

If *side* = 'L', $m \geq k \geq 0$

If *side* = 'R', $n \geq k \geq 0$.

a (local)

COMPLEX for pcunmrq

DOUBLE COMPLEX for pzunmrq.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R'. The i-th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gerqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

tau (local)

COMPLEX for pcunmrq

DOUBLE COMPLEX for pzunmrq

Array of size *LOCc*(*ja*+*k*-1).

Contains the scalar factor *tau*(i) of elementary reflectors $H(i)$ as returned by p?gerqf. *tau* is tied to the distributed matrix A.

c (local)

COMPLEX for pcunmrq

DOUBLE COMPLEX for pzunmrq.

Pointer into the local memory to an array of local size (*lld_c*, *LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(C) to be factored.

ic, *jc*

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

COMPLEX for pcunmrq

DOUBLE COMPLEX for pzunmrq.

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

If *side* = 'L',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0+numroc(numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* = 'R',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a***mb_a*

end if

where

lcmp = *lcm*/NPROW with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q** sub(*C*) or *Q*'*sub (*C*), or sub(*C*)* *Q*', or sub(*C*)* *Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```
call pstzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdtzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pctzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pztzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The p?tzrzfroutine reduces the *m*-by-*n* (*m* ≤ *n*) real/complex upper trapezoidal matrix sub(*A*)= *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix *A* is factored as

A = (*R* 0)**Z*,

where *Z* is an *n*-by-*n* orthogonal/unitary matrix and *R* is an *m*-by-*m* upper triangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows in the matrix sub(*A*); (*m*≥0).
n	(global) INTEGER. The number of columns in the matrix sub(*A*) (*n*≥0).
a	(local) REAL for pstzrzf DOUBLE PRECISION for pdtzrzf. COMPLEX for pctzrzf. DOUBLE COMPLEX for pztzrzf.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)). Contains the local pieces of the *m*-by-*n* distributed matrix sub (*A*) to be factored.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work

(local)

REAL for pstzrzf

DOUBLE PRECISION for pdtzrzf.

COMPLEX for pctzrzf.

DOUBLE COMPLEX for pztzrzf.

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least *lwork*≥*mb_a**(*mp*0+*nq*0+*mb_a*), where

iroff = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc (*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc (*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, the leading *m*-by-*m* upper triangular part of sub(*A*) contains the upper triangular matrix *R*, and elements *m*+1 to *n* of the first *m* rows of sub (*A*), with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

tau

(local)

REAL for pstzrzf

DOUBLE PRECISION for pdtzrzf.

COMPLEX for pctzrzf.

DOUBLE COMPLEX for pztzrzf.

Array of size *LOCr*(*ia*+*m*-1).

Contains the scalar factor of elementary reflectors. *tau* is tied to the distributed matrix *A*.

info (global) INTEGER.

= 0: the execution is successful.

< 0:if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The factorization is obtained by the Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is or whose conjugate transpose is used to introduce zeros into the (*m* - *k* +1)-th row of sub(*A*), is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where

T(*k*) = *i* - *tau***u*(*k*)**u*(*k*)',

$$u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and *Z*(*k*) is an (*n* - *m*) element vector. *tau* and *Z*(*k*) are chosen to annihilate the elements of the *k*-th row of sub(*A*). The scalar *tau* is returned in the *k*-th element of *tau* and the vector *u*(*k*) in the *k*-th row of sub(*A*), such that the elements of *Z*(*k*) are in *a*(*k*, *m* + 1),..., *a*(*k*, *n*). The elements of *R* are returned in the upper triangular part of sub(*A*). *Z* is given by

Z = *Z*(1) * *Z*(2) *... * *Z*(*m*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormrz

Multiplies a general matrix by the orthogonal matrix from a reduction to upper triangular form formed by p?tzrzf*.*

Syntax

```
call psormrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

```
call pdormrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files

Description

This routine overwrites the general real *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^T*sub(*C*)	sub(*C*)*Q^T

where *Q* is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1) *H*(2)... *H*(*k*)

as returned by p?tzrzf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER

='L': *Q* or Q^T is applied from the left.

='R': *Q* or Q^T is applied from the right.

trans (global) CHARACTER

='N', no transpose, *Q* is applied.

='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m* ≥ *k* ≥0

If *side* = 'R', *n* ≥ *k* ≥0.

l (global)

The columns of the distributed matrix sub(*A*) containing the meaningful part of the Householder reflectors.

If *side* = 'L', *m* ≥ *l* ≥0

If *side* = 'R', *n* ≥ *l* ≥0.

a (local)

REAL for psormrz

DOUBLE PRECISION for pdormrz.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*,*LOCc*(*ja*+*n*-1)) if *side* = 'R', where *lld_a* ≥ max(1,*LOCr*(*ia*+*k*-1)).

The *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*-1, as returned by p?tzrzf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1, *ja*:*). *A*(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau

(local)

REAL for psormrz

DOUBLE PRECISION for pdormrz

Array of size *LOCc*(*ia*+*k*-1).

Contains the scalar factor *tau*(*i*) of elementary reflectors *H*(*i*) as returned by p?tzrzf. *tau* is tied to the distributed matrix *A*.

c

(local)

REAL for psormrz

DOUBLE PRECISION for pdormrz

Pointer into the local memory to an array of local size (*lld_c*,*LOCc*(*jc* +*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc*

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

REAL for psormrz

DOUBLE PRECISION for pdormrz.

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

If *side* = 'L',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0 + numroc(numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))*mb_a) + *mb_a**mb_a

else if *side* ='R',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)*mb_a) + *mb_a**mb_a

end if

where

lcmp = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q**sub(*C*), or *Q*'*sub (*C*), or sub(*C*)**Q*', or sub(*C*)**Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmrz

Multiplies a general matrix by the unitary transformation matrix from a reduction to upper triangular form determined by p?tzrzf.

Syntax

call pcunmrz(*side*, *trans*, *m*, *n*, *k*, *l*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmrz(*side*, *trans*, *m*, *n*, *k*, *l*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Include Files

Description

This routine overwrites the general complex *m*-by-*n* distributed matrix sub (*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1)' *H*(2)'... *H*(*k*)'

as returned by pctzrzf/pztzrzf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER

='L': *Q* or Q^H is applied from the left.

='R': *Q* or Q^H is applied from the right.

trans (global) CHARACTER

='N', no transpose, *Q* is applied.

='C', conjugate transpose, Q^H is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*), (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*), (*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m*≥*k*≥0

If *side* = 'R', *n*≥*k*≥0.

l (global) INTEGER. The columns of the distributed matrix sub(*A*) containing the meaningful part of the Householder reflectors.

If *side* = 'L', *m*≥*l*≥0

If *side* = 'R', *n*≥*l*≥0.

a (local)

COMPLEX for pcunmrz

DOUBLE COMPLEX for pzunmrz.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*,*LOCc*(*ja*+*n*-1)) if *side* = 'R', where *lld_a* ≥ max(1, *LOCr*(*ja*+*k*-1)). The *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1, *ja*:*). *A*(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau

(local)

COMPLEX for pcunmrz

DOUBLE COMPLEX for pzunmrz

Array of size *LOCc*(*ia*+*k*-1).

Contains the scalar factor *tau*(*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

c

(local)

COMPLEX for pcunmrz

DOUBLE COMPLEX for pzunmrz.

Pointer into the local memory to an array of local size (*lld_c*,*LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc*

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

COMPLEX for pcunmrz

DOUBLE COMPLEX for pzunmrz.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

If *side* = 'L',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0+max(*mqa*0+numroc(numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* ='R',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0+*nqc*0)**mb_a*) + *mb_a***mb_a*

end if

where

lcmp = *lcm*/NPROW with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q** sub(*C*), or *Q*'*sub (*C*), or sub(*C*)**Q*', or sub(*C*)**Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ggqrf

Computes the generalized QR factorization.

Syntax

call psggqrf(*n*, *m*, *p*, *a*, *ia*, *ja*, *desca*, *taua*, *b*, *ib*, *jb*, *descb*, *taub*, *work*, *lwork*, *info*)

```
call pdggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
call pcggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
call pzggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

Include Files

Description

The p?ggqrfroutine forms the generalized *QR* factorization of an *n*-by-*m* matrix

sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*m*-1)

and an *n*-by-*p* matrix

sub(*B*) = *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*p*-1):

as

sub(A) = *Q***R*, sub(*B*) = *Q***T***Z*,

where *Q* is an *n*-by-*n* orthogonal/unitary matrix, *Z* is a *p*-by-*p* orthogonal/unitary matrix, and *R* and *T* assume one of the forms:

If $n \geq m$

$$R = \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \begin{matrix} m \\ n-m \end{matrix}$$
$$m$$

or if $n < m$

$$R = \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} \begin{matrix} n \\ \end{matrix}$$
$$n \quad m-n$$

where R_{11} is upper triangular, and

$$T = \begin{pmatrix} 0 & T_{12} \end{pmatrix} \begin{matrix} n, \text{ if } n \leq p, \\ \end{matrix}$$
$$p-n \quad n$$

$$\text{or } T = \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \begin{pmatrix} n-p \\ p \end{pmatrix}, \text{ if } n > p,$$
$$p$$

where T_{12} or T_{21} is an upper triangular matrix.

In particular, if sub(*B*) is square and nonsingular, the *GQR* factorization of sub(*A*) and sub(*B*) implicitly gives the *QR* factorization of inv (sub(*B*))* sub (*A*):

inv(sub(*B*))*sub(*A*) = Z^H*(inv(*T*)**R*)

Input Parameters

n (global) INTEGER. The number of rows in the distributed matrices sub (*A*) and sub(*B*) (*n*≥0).

m (global) INTEGER. The number of columns in the distributed matrix sub(*A*) (*m*≥0).

p INTEGER. The number of columns in the distributed matrix sub(*B*) (*p*≥0).

a (local)

REAL for psggqrf

DOUBLE PRECISION for pdggqrf

COMPLEX for pcggqrf

DOUBLE COMPLEX for pzggqrf.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)). Contains the local pieces of the *n*-by-*m* matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

b (local)

REAL for psggqrf

DOUBLE PRECISION for pdggqrf

COMPLEX for pcggqrf

DOUBLE COMPLEX for pzggqrf.

Pointer into the local memory to an array of size (*lld_b*,*LOCc*(*jb*+*p*-1)). Contains the local pieces of the *n*-by-*p* matrix sub(*B*) to be factored.

ib, *jb* (global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix B.

work (local)

REAL for psggqrf

DOUBLE PRECISION for pdggqrf

COMPLEX for pcggqrf

DOUBLE COMPLEX for pzggqrf.

Workspace array of size of *lwork*.

lwork (local or global) INTEGER. Sze of *work*, must be at least

lwork≥max(*nb_a**(*npa*0+*mqa*0+*nb_a*), max((*nb_a**(*nb_a*-1))/2, (*pqb*0+*npb*0)**nb_a*)+*nb_a***nb_a*, *mb_b**(*npb*0+*pqb*0+*mb_b*)),

where

iroffa = mod(*ia*-1, *mb_A*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

npa0 = numroc (*n*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

mqa0 = numroc (*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

iroffb = mod(*ib*-1, *mb_b*),

icoffb = mod(*jb*-1, *nb_b*),

ibrow = indxg2p(*ib*, *mb_b*, MYROW, *rsrc_b*, NPROW),

ibcol = indxg2p(*jb*, *nb_b*, MYCOL, *csrc_b*, NPCOL),

npb0 = numroc (*n*+*iroffa*, *mb_b*, MYROW, *Ibrow*, NPROW),

pqb0 = numroc(*m*+*icoffb*, *nb_b*, MYCOL, *ibcol*, NPCOL)

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, the elements on and above the diagonal of sub (*A*) contain the min(*n*, *m*)-by-*m* upper trapezoidal matrix *R* (*R* is upper triangular if *n*≥*m*); the elements below the diagonal, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*, *m*) elementary reflectors. (See Application Notes below).

taua, *taub*

(local)

REAL for psggqrf

DOUBLE PRECISION for pdggqrf

COMPLEX for pcggqrf

DOUBLE COMPLEX for pzggqrf.

Arrays of size *LOCc*(*ja*+min(*n*,*m*)-1) for *taua* and *LOCr*(*ib*+*n*-1) for *taub*.

The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. *taua* is tied to the distributed matrix *A*. (See Application Notes below).

	The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*. *taub* is tied to the distributed matrix *B*. (See Application Notes below).
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ja*)**H*(*ja*+1)*...**H*(*ja*+*k*-1),

where *k*= min(*n*,*m*).

Each *H*(*i*) has the form

H(*i*) = *i* - *taua***v***v'*

where *taua* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*-1) = 0 and *v*(*i*) = 1; v(*i*+1:*n*) is stored on exit in *A*(*ia*+*i*:*ia*+*n*-1, *ja*+*i*-1) , and *taua* in *taua*(*ja*+*i*-1).To form *Q* explicitly, use ScaLAPACK subroutine p?orgqr/p?ungqr. To use *Q* to update another matrix, use ScaLAPACK subroutine p?ormqr/p?unmqr.

The matrix *Z* is represented as a product of elementary reflectors

Z = *H*(*ib*)**H*(*ib*+1)*...**H*(*ib*+*k*-1), where *k*= min(*n*,*p*).

Each *H*(*i*) has the form

H(*i*) = *i* - *taub***v***v'*

where *taub* is a real/complex scalar, and *v* is a real/complex vector with *v*(*p*-*k*+*i*+1:*p*) = 0 and *v*(*p*-*k*+*i*) = 1; *v*(1:*p*-*k*+*i*-1) is stored on exit in *B*(*ib*+*n*-*k*+*i*-1,*jb*:*jb*+*p*-*k*+*i*-2), and *taub* in *taub*(*ib*+*n*-*k*+*i*-1). To form *Z* explicitly, use ScaLAPACK subroutine p?orgrq/p?ungrq. To use *Z* to update another matrix, use ScaLAPACK subroutine p?ormrq/p?unmrq.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ggrqf

Computes the generalized RQ factorization.

Syntax

```
call psggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
call pdggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
call pcggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
call pzggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

Include Files

Description

The p?ggrqfroutine forms the generalized *RQ* factorization of an *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) and a *p*-by-*n* matrix sub(*B*) = *B*(*ib*:*ib*+*p*-1, *jb*:*jb*+*n*-1):

sub(*A*) = *R***Q*, sub(*B*) = *Z***T***Q*,

where *Q* is an *n*-by-*n* orthogonal/unitary matrix, *Z* is a *p*-by-*p* orthogonal/unitary matrix, and *R* and *T* assume one of the forms:

$$R = \begin{matrix} m \\ \\ \end{matrix} \begin{matrix} \begin{pmatrix} 0 & R_{12} \end{pmatrix}, \text{ if } m \le n, \\ n-m \quad m \end{matrix}$$

or

$$R = \begin{matrix} \begin{pmatrix} R_{11} \\ R_{12} \end{pmatrix} \begin{matrix} m-n \\ n \end{matrix} \\ n \end{matrix}, \text{ if } m > n$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{matrix} \begin{pmatrix} T_{11} \\ 0 \end{pmatrix} \begin{matrix} n \\ p-n \end{matrix} \\ n \end{matrix}, \text{ if } p \ge n$$

or

$$T = \begin{matrix} p \\ \\ \end{matrix} \begin{matrix} \begin{pmatrix} T_{11} & T_{12} \end{pmatrix}, \text{ if } p < n, \\ p \quad n-p \end{matrix}$$

where T_{11} is upper triangular.

In particular, if sub(*B*) is square and nonsingular, the *GRQ* factorization of sub(*A*) and sub(*B*) implicitly gives the *RQ* factorization of sub (*A*)*inv(sub(*B*)):

```
sub(A)*inv(sub(B))= (R*inv(T))*Z'
```

where inv(sub(*B*)) denotes the inverse of the matrix sub(*B*), and *Z*′ denotes the transpose (conjugate transpose) of matrix *Z*.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrices sub (*A*) (*m*≥0).
p	INTEGER. The number of rows in the distributed matrix sub(*B*) (*p*≥0).
n	(global) INTEGER. The number of columns in the distributed matrices sub(*A*) and sub(*B*) (*n*≥0).
a	(local)

REAL for psggrqf

DOUBLE PRECISION for pdggrqf

COMPLEX for pcggrqf

DOUBLE COMPLEX for pzggrqf.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+n-1)). Contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

b (local)

REAL for psggrqf

DOUBLE PRECISION for pdggrqf

COMPLEX for pcggrqf

DOUBLE COMPLEX for pzggrqf.

Pointer into the local memory to an array of size (*lld_b*,*LOCc*(*jb*+*n*-1)).

Contains the local pieces of the *p*-by-*n* matrix sub(*B*) to be factored.

ib, *jb* (global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

work (local)

REAL for psggrqf

DOUBLE PRECISION for pdggrqf

COMPLEX for pcggrqf

DOUBLE COMPLEX for pzggrqf.

Workspace array of size of *lwork*.

lwork (local or global) INTEGER.

Size of *work*, must be at least *lwork*≥max(*mb_a**(*mpa*0+*nqa*0+*mb_a*), max((*mb_a**(*mb_a*-1))/2, (*ppb*0+*nqb*0)**mb_a*) + *mb_a***mb_a*, *nb_b**(*ppb*0+*nqb*0+*nb_b*)), where

iroffa = mod(*ia*-1, *mb_A*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

mpa0 = *numroc* (*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

nqa0 = *numroc* (*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

iroffb = mod(*ib*-*1*, *mb_b*),

icoffb = mod(*jb*-*1*, *nb_b*),

ibrow = indxg2p(*ib*, *mb_b*, MYROW, *rsrc_b*, NPROW),

ibcol = indxg2p(*jb*, *nb_b*, MYCOL, *csrc_b*, NPCOL),

ppb0 = numroc (*p*+*iroffb*, *mb_b*, MYROW, *ibrow*,NPROW),

nqb0 = numroc (*n*+*icoffb*, *nb_b*, MYCOL, *ibcol*,NPCOL)

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, if *m*≤*n*, the upper triangle of *A*(*ia*:*ia*+*m*-1, *ja*+*n*-*m*:*ja*+*n*-1) contains the *m*-by-*m* upper triangular matrix *R*; if *m*≥*n*, the elements on and above the (*m*-*n*)-th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*; the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors (see *Application Notes* below).

taua, *taub*

(local)

REAL for psggqrf

DOUBLE PRECISION for pdggqrf

COMPLEX for pcggqrf

DOUBLE COMPLEX for pzggqrf.

Arrays of size *LOCr*(*ia*+*m*-1)for *taua* and *LOCc*(*jb*+min(p,*n*)-1) for *taub*.

The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. *taua* is tied to the distributed matrix *A*.(See *Application Notes* below).

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*. *taub* is tied to the distributed matrix *B*. (See *Application Notes* below).

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ia*)**H*(*ia*+1)*...**H*(*ia*+*k*-1),

where *k*= min(*m*,*n*).

Each *H*(*i*) has the form

H(*i*) = *i* - *taua***v***v*'

where *taua* is a real/complex scalar, and *v* is a real/complex vector with *v*(*n*-*k*+*i*+1:*n*) = 0 and *v*(*n*-*k*+*i*) = 1; v(1:*n*-*k*+*i*-1) is stored on exit in *A*(*ia*+*m*-*k*+*i*-1, *ja*:*ja*+*n*-*k*+*i*-2), and *taua* in *taua*(*ia*+*m*-*k*+*i*-1). To form *Q* explicitly, use ScaLAPACK subroutine p?orgrq/p?ungrq. To use *Q* to update another matrix, use ScaLAPACK subroutine p?ormrq/p?unmrq.

The matrix *Z* is represented as a product of elementary reflectors

Z = *H*(*jb*)**H*(*jb*+1)*...**H*(*jb*+*k*-1), where *k*= min(*p*,*n*).

Each *H*(*i*) has the form

H(*i*) = *i* - *taub***v***v*'

where *taub* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*-1) = 0 and *v*(*i*)= 1; *v*(*i*+1:*p*) is stored on exit in *B*(*ib*+*i*:*ib*+*p*-1,*jb*+*i*-1), and *taub* in *taub*(*jb*+*i*-1). To form *Z* explicitly, use ScaLAPACK subroutine p?orgqr/p?ungqr. To use *Z* to update another matrix, use ScaLAPACK subroutine p?ormqr/p?unmqr.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Symmetric Eigenvalue Problems: ScaLAPACK Computational Routines

To solve a symmetric eigenproblem with ScaLAPACK, you usually need to reduce the matrix to real tridiagonal form *T* and then find the eigenvalues and eigenvectors of the tridiagonal matrix *T*. ScaLAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in Table "Computational Routines for Solving Symmetric Eigenproblems".

There are different routines for symmetric eigenproblems, depending on whether you need eigenvalues only or eigenvectors as well, and on the algorithm used (either the *QTQ* algorithm, or bisection followed by inverse iteration).

Computational Routines for Solving Symmetric Eigenproblems

Operation	Dense symmetric/ Hermitian matrix	Orthogonal/unitary matrix	Symmetric tridiagonal matrix
Reduce to tridiagonal form $A = QTQ^H$	p?sytrd/p?hetrd		
Multiply matrix after reduction		p?ormtr/p?unmtr	
Find all eigenvalues and eigenvectors of a tridiagonal matrix *T* by a *QTQ* method			steqr2*
Find selected eigenvalues of a tridiagonal matrix *T* via bisection			p?stebz

Operation	Dense symmetric/ Hermitian matrix	Orthogonal/unitary matrix	Symmetric tridiagonal matrix
Find selected eigenvectors of a tridiagonal matrix *T* by inverse iteration			p?stein

* This routine is described as part of auxiliary ScaLAPACK routines.

p?syngst

Reduces a complex Hermitian-definite generalized eigenproblem to standard form.

Syntax

```
call pssyngst (ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, work, lwork, info )
```

```
call pdsyngst (ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, work, lwork, info )
```

Description

p?syngst reduces a complex Hermitian-definite generalized eigenproblem to standard form.

p?syngst performs the same function as p?hegst, but is based on rank 2K updates, which are faster and more scalable than triangular solves (the basis of p?syngst).

p?syngst calls p?hegst when *uplo*='U', hence p?hengst provides improved performance only when *uplo*='L', *ibtype*=1.

p?syngst also calls p?hegst when insufficient workspace is provided, hence p?syngst provides improved performance only when *lwork* >= 2 * NP0 * NB + NQ0 * NB + NB * NB

In the following sub(*A*) denotes *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*n*-1, *jb*:*jb* +*n*-1).

If *ibtype* = 1, the problem is sub(*A*)*x = lambda*sub(*B*)*x, and sub(*A*) is overwritten by inv(U^H)*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(L^H)

If *ibtype* = 2 or 3, the problem is sub(*A*)*sub(*B*)*x = lambda*x or sub(*B*)*sub(*A*)*x = lambda*x, and sub(*A*) is overwritten by *U**sub(*A*)*U^H or L^H*sub(*A*)**L*.

sub(*B*) must have been previously factorized as U^H**U* or *L*L^H by p?potrf.

Input Parameters

ibtype (global)

INTEGER.

= 1: compute inv(U^H)*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(L^H);

= 2 or 3: compute *U*sub(*A*)*U^H or L^H*sub(*A*)**L*.

uplo (global)

CHARACTER.

= 'U': Upper triangle of sub(*A*) is stored and sub(*B*) is factored as U^H**U*;

= 'L': Lower triangle of sub(*A*) is stored and sub(*B*) is factored as *L*L^H.

n (global)

INTEGER.

The order of the matrices sub(*A*) and sub(*B*). *n* >= 0.

a (local)

REAL for pssyngst

DOUBLE PRECISION for pdsyngst

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia (global)

INTEGER.

A's global row index, which points to the beginning of the submatrix which is to be operated on.

ja (global)

INTEGER.

A's global column index, which points to the beginning of the submatrix which is to be operated on.

desca (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

b (local)

REAL for pssyngst

DOUBLE PRECISION for pdsyngst

Pointer into the local memory to an array of size (*lld_b*,*LOCc*(*jb*+*n*-1)).

On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of sub(*B*), as returned by p?potrf.

ib (global)

INTEGER.

B's global row index, which points to the beginning of the submatrix which is to be operated on.

jb (global)

INTEGER.

B's global column index, which points to the beginning of the submatrix which is to be operated on.

descb (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *B*.

work (local)

REAL for pssyngst

DOUBLE PRECISION for pdsyngst

Array, size (*lwork*)

lwork (local or global)

INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork* >= MAX(NB * (NP0 +1), 3 * NB)

When *ibtype* = 1 and *uplo* = 'L', p?syngst provides improved performance when *lwork* >= 2 * NP0 * NB + NQ0 * NB + NB * NB,

where NB = *mb_a* = *nb_a*,

NP0 = numroc(*n*, NB, 0, 0, NPROW),

NQ0 = numroc(*n*, NB, 0, 0, NPROW),

numroc is a ScaLAPACK tool functions

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, if *info* = 0, the transformed matrix, stored in the same format as sub(*A*).

scale (global)

REAL for pssyngst

DOUBLE PRECISION for pdsyngst

Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, *scale* is always returned as 1.0, it is returned here to allow for future enhancement.

work (local)

REAL for pssyngst

DOUBLE PRECISION for pdsyngst

Array, size (*lwork*)

On exit, `work(1)` returns the minimal and optimal `lwork`.

`info` (global)

INTEGER.

= 0: successful exit

< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then `info` = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then `info` = -*i*.

p?syntrd

Reduces a real symmetric matrix to symmetric tridiagonal form.

Syntax

```
call pssyntrd (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info )
call pdsyntrd (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info )
```

Description

`p?syntrd` is a prototype version of `p?sytrd` which uses tailored codes (either the serial, `?sytrd`, or the parallel code, `p?syttrd`) when the workspace provided by the user is adequate.

`p?syntrd` reduces a real symmetric matrix sub(*A*) to symmetric tridiagonal form *T* by an orthogonal similarity transformation:

Q' * sub(*A*) * *Q* = *T*, where sub(*A*) = *A*(`ia`:`ia`+`n`-1,`ja`:`ja`+`n`-1).

Features

`p?syntrd` is faster than `p?sytrd` on almost all matrices, particularly small ones (i.e. `n` < 500 * sqrt(P)), provided that enough workspace is available to use the tailored codes.

The tailored codes provide performance that is essentially independent of the input data layout.

The tailored codes place no restrictions on `ia`, `ja`, MB or NB. At present, `ia`, `ja`, MB and NB are restricted to those values allowed by `p?hetrd` to keep the interface simple (see the Application Notes section for more information about the restrictions).

Input Parameters

`uplo` (global)

CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is stored:

= 'U': Upper triangular

= 'L': Lower triangular

`n` (global)

INTEGER.

The number of rows and columns to be operated on, i.e. the order of the distributed submatrix sub(*A*). `n` >= 0.

`a` (local)

REAL for pssyntrd

DOUBLE PRECISION for pdsyntrd

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the symmetric distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia

(global)

INTEGER.

The row index in the global array *a* indicating the first row of sub(*A*).

ja

(global)

INTEGER.

The column index in the global array *a* indicating the first column of sub(*A*).

desca

(global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

work

(local)

REAL for pssyntrd

DOUBLE PRECISION for pdsyntrd

Array, size (*lwork*)

lwork

(local or global)

INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork* >= MAX(NB * (NP +1), 3 * NB)

For optimal performance, greater workspace is needed, i.e.

lwork >= 2*(*ANB*+1)*(4**NPS*+2) + (*NPS* + 4) * *NPS*

ANB = pjlaenv(*ICTXT*, 3, 'p?syttrd', 'L', 0, 0, 0, 0)

ICTXT = *desca*(*ctxt_*)

SQNPC = INT(sqrt(REAL(*NPROW* * *NPCOL*)))

numroc is a ScaLAPACK tool function.

pjlaenv is a ScaLAPACK environmental inquiry function.

NPROW and *NPCOL* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

a

On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors; if *uplo* = 'L', the diagonal and first subdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors. See **Further Details**.

d

(local)

`REAL` for `pssyntrd`

`DOUBLE PRECISION` for `pdsyntrd`

Array, size LOCc(*ja*+*n*-1)

The diagonal elements of the tridiagonal matrix *T*: *d*(i) = *A*(i,i). *d* is tied to the distributed matrix *A*.

e

(local)

`REAL` for `pssyntrd`

`DOUBLE PRECISION` for `pdsyntrd`

Array, size LOCc(*ja*+*n*-1) if *uplo* = 'U', LOCc(*ja*+*n*-2) otherwise.

The off-diagonal elements of the tridiagonal matrix *T*: *e*(i) = *A*(i,i+1) if *uplo* = 'U', *e*(i) = *A*(i+1,i) if *uplo* = 'L'. *e* is tied to the distributed matrix *A*.

tau

(local)

`REAL` for `pssyntrd`

`DOUBLE PRECISION` for `pdsyntrd`

Array, size LOCc(*ja*+*n*-1).

This array contains the scalar factors *tau* of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work

(local)

`REAL` for `pssyntrd`

`DOUBLE PRECISION` for `pdsyntrd`

Array, size (*lwork*)

On exit, *work*`(1)` returns the optimal *lwork*.

info

(global)

INTEGER.

= 0: successful exit

< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

Q = H(*n*-1) . . . H(2) H(1).

Each H(i) has the form

H(i) = I - *tau* * v * v', where *tau* is a complex scalar, and v is a complex vector with v(i+1:*n*) = 0 and v(i) = 1; v(1:i-1) is stored on exit in *A*(*ia*:*ia*+i-2,*ja*+i), and *tau* in *tau*(*ja*+i-1).

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

Q = H(1) H(2) . . . H(*n*-1).

Each H(i) has the form

H(i) = I - *tau* * v * v', where *tau* is a complex scalar, and v is a complex vector with v(1:i) = 0 and v(i+1) = 1; v(i+2:*n*) is stored on exit in *A*(*ia*+i+1:*ia*+*n*-1,*ja*+i-1), and *tau* in *tau*(*ja*+i-1).

The contents of sub(*A*) on exit are illustrated by the following examples with *n* = 5:

if *uplo* = 'U':

$$\begin{pmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v3 \\ & & & d & e \\ & & & & d \end{pmatrix}$$

if *uplo* = 'L':

$$\begin{pmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{pmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *vi* denotes an element of the vector defining H(*i*).

Alignment requirements

The distributed submatrix sub(*A*) must verify some alignment properties, namely the following expression should be true:

(*mb_a* = *nb_a* and *IROFFA* = *ICOFFA* and *IROFFA* = 0) with *IROFFA* = mod(*ia*-1, *mb_a*), and *ICOFFA* = mod(*ja*-1, *nb_a*).

p?sytrd

Reduces a symmetric matrix to real symmetric tridiagonal form by an orthogonal similarity transformation.

Syntax

```
call pssytrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

```
call pdsytrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Include Files

Description

The p?sytrd routine reduces a real symmetric matrix sub(*A*) to symmetric tridiagonal form *T* by an orthogonal similarity transformation:

Q'*sub(*A*)**Q* = *T*,

where sub(*A*) = *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1).

Input Parameters

uplo (global) CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is stored:

If *uplo* = 'U', upper triangular

If *uplo* = 'L', lower triangular

n (global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).

a (local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the symmetric distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. See *Application Notes* below.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work (local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Workspace array of size *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least:

lwork ≥ max(NB*(*np* +1), 3*NB),

where NB = *mb_a* = *nb_a*,

np = numroc(*n*, NB, MYROW, *iarow*, NPROW),

iarow = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW).

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors; if *uplo* = 'L', the diagonal and first subdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors. See *Application Notes* below.

d

(local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Arrays of size *LOCc*(*ja*+*n*-1) .The diagonal elements of the tridiagonal matrix *T*:

d(*i*)= *A*(*i*,*i*).

d is tied to the distributed matrix *A*.

e

(local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Arrays of size *LOCc*(*ja*+*n*-1) if *uplo* = 'U', *LOCc*(*ja*+*n*-2) otherwise.

The off-diagonal elements of the tridiagonal matrix *T*:

e(*i*)= *A*(*i*,*i*+1) if *uplo* = 'U',

e(*i*) = *A*(*i*+1,*i*) if *uplo* = 'L'.

e is tied to the distributed matrix *A*.

tau

(local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Arrays of size *LOCc*(*ja*+*n*-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*n*-1)... *H*(2) *H*(1).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau* * *v* * *v*',

where *tau* is a real scalar, and *v* is a real vector with *v*(*i*+1:*n*) = 0 and *v*(*i*) = 1; *v*(1:*i*-1) is stored on exit in *A*(*ia*:*ia*+*i*-2, *ja*+*i*), and *tau* in *tau*(*ja*+*i*-1).

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1) *H*(2)... *H*(*n*-1).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau* * *v* * *v*',

where *tau* is a real scalar, and *v* is a real vector with *v*(1:*i*) = 0 and *v*(*i*+1) = 1; *v*(*i*+2:*n*) is stored on exit in *A*(*ia*+*i*+1:*ia*+*n*-1,*ja*+*i*-1), and *tau* in *tau*(*ja*+*i*-1).

The contents of sub(*A*) on exit are illustrated by the following examples with *n* = 5:

If *uplo* = 'U':

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If *uplo* = 'L':

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *vi* denotes an element of the vector defining *H*(*i*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormtr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to tridiagonal form determined by `p?sytrd`.

Syntax

`call psormtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)`

`call pdormtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)`

Include Files

Description

This routine overwrites the general real distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^T*sub(*C*)	sub(*C*)*Q^T

where *Q* is a real orthogonal distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'.

Q is defined as the product of *nq* elementary reflectors, as returned by `p?sytrd`.

If *uplo* = 'U', *Q* = *H*(*nq*-1)... *H*(2) *H*(1);

If *uplo* = 'L', *Q* = *H*(1) *H*(2)... *H*(*nq*-1).

Input Parameters

side (global) CHARACTER

='L': *Q* or Q^T is applied from the left.

='R': *Q* or Q^T is applied from the right.

trans (global) CHARACTER

='N', no transpose, *Q* is applied.

='T', transpose, Q^T is applied.

uplo (global) CHARACTER.

= 'U': Upper triangle of *A*(*ia*:*, *ja*:*) contains elementary reflectors from `p?sytrd`;

= 'L': Lower triangle of *A*(*ia*:*,*ja*:*) contains elementary reflectors from `p?sytrd`

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

a (local)

REAL for psormtr

DOUBLE PRECISION for pdormtr.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*,*LOCc*(*ja*+*n*-1)) if *side* = 'R'.

Contains the vectors that define the elementary reflectors, as returned by p?sytrd.

If *side*='L', *lld_a* ≥ max(1,*LOCr*(*ia*+*m*-1));

If *side* ='R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau

(local)

REAL for psormtr

DOUBLE PRECISION for pdormtr.

Array of size of *ltau* where

if *side* = 'L' and *uplo* = 'U', *ltau* = *LOCc*(*m_a*),

if *side* = 'L' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*m*-2),

if *side* = 'R' and *uplo* = 'U', *ltau* = *LOCc*(*n_a*),

if *side* = 'R' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*n*-2).

tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by p?sytrd. *tau* is tied to the distributed matrix *A*.

c

(local) REAL for psormtr

DOUBLE PRECISION for pdormtr.

Pointer into the local memory to an array of size (*lld_c*,*LOCc*(*jc*+*n*-1)). Contains the local pieces of the distributed matrix sub (*C*).

ic, *jc*

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

REAL for psormtr

DOUBLE PRECISION for pdormtr.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

if *uplo* = 'U',

iaa= *ia*; *jaa*= *ja*+1, *icc*= *ic*; *jcc*= *jc*;

else *uplo* = 'L',

iaa= *ia*+1, *jaa*= *ja*;

If *side* = 'L',

icc= *ic*+1; *jcc*= *jc*;

else *icc*= *ic*; *jcc*= *jc*+1;

end if

end if

If *side* = 'L',

mi= *m*-1; *ni*= *n*

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a***nb_a*

else

If *side* = 'R',

mi= *m*; *mi* = *n*-1;

lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0+numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*)+ *nb_a***nb_a*

end if

where *lcmq* = *lcm*/NPCOL with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),

icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q**sub(*C*), or *Q'**sub(*C*), or sub(*C*)**Q'*, or sub(*C*)**Q*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?hengst

Reduces a complex Hermitian-definite generalized eigenproblem to standard form.

Syntax

```
call pchengst (ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, work, lwork, info )
```

```
call pzhengst (ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, work, lwork, info )
```

Description

p?hengst reduces a complex Hermitian-definite generalized eigenproblem to standard form.

p?hengst performs the same function as p?hegst, but is based on rank 2K updates, which are faster and more scalable than triangular solves (the basis of p?hengst).

p?hengst calls p?hegst when *uplo*='U', hence p?hengst provides improved performance only when *uplo*='L' and *ibtype*=1.

p?hengst also calls p?hegst when insufficient workspace is provided, hence p?hengst provides improved performance only when *lwork* is sufficient (as described in the parameter descriptions).

In the following sub(*A*) denotes the submatrix *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(*B*) denotes the submatrix *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1).

If *ibtype* = 1, the problem is sub(*A*)*x = lambda*sub(*B*)*x, and sub(*A*) is overwritten by inv(U^H)*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(L^H)

If *ibtype* = 2 or 3, the problem is sub(*A*)*sub(*B*)*x = lambda*x or sub(*B*)*sub(*A*)*x = lambda*x, and sub(*A*) is overwritten by *U**sub(*A*)*U^H or L^H*sub(*A*)**L*.

sub(*B*) must have been previously factorized as U^H**U* or *L**L^H by p?potrf.

Input Parameters

ibtype	(global) INTEGER. = 1: compute inv(U^H)*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(L^H);

= 2 or 3: compute U*sub(A)*U^H or L^H*sub(A)*L.

uplo (global)

CHARACTER.

= 'U': Upper triangle of sub(A) is stored and sub(B) is factored as U^H*U;

= 'L': Lower triangle of sub(A) is stored and sub(B) is factored as $L*L^H$.

n (global)

INTEGER.

The order of the matrices sub(A) and sub(B). *n* >= 0.

a (local)

`COMPLEX` for `pchengst`

`DOUBLE COMPLEX` for `pzhengst`

Pointer into the local memory to an array of size `(lld_a,LOCc(ja+n-1))`.

On entry, this array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(A). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(A) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(A) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia (global)

INTEGER.

Global row index of matrix A, which points to the beginning of the submatrix on which to operate.

ja (global)

INTEGER.

Global column index of matrix A, which points to the beginning of the submatrix on which to operate.

desca (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix A.

b (local)

`COMPLEX` for `pchengst`

`DOUBLE COMPLEX` for `pzhengst`

Pointer into the local memory to an array of size `(lld_b,LOCc(jb+n-1))`.

ib (global)

INTEGER.

Global row index of matrix *B*, which points to the beginning of the submatrix on which to operate.

jb

(global)

INTEGER.

Global column index of matrix *B*, which points to the beginning of the submatrix on which to operate.

descb

(global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *B*.

work

(local)

`COMPLEX` for `pchengst`

`DOUBLE COMPLEX` for `pzhengst`

Array, size (*lwork*)

On exit, *work*(1) returns the minimal and optimal *lwork*.

lwork

(local)

INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork* >= MAX(NB * (NP0 +1), 3 * NB).

When *ibtype* = 1 and *uplo* = 'L', `p?hengst` provides improved performance when *lwork* >= 2 * *NP0* * *NB* + *NQ0* * *NB* + *NB* * *NB*, where *NB* = *mb_a* = *nb_a*, *NP0* = `numroc`(*n*, *NB*, 0, 0, `NPROW`), *NQ0* = `numroc`(*n*, *NB*, 0, 0, `NPROW`), and `numroc` is a ScaLAPACK tool function.

`MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

On exit, if *info* = 0, the transformed matrix, stored in the same format as sub(*A*).

scale

(global)

`REAL` for `pchengst`

`DOUBLE PRECISION` for `pzhengst`

Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine.

scale is always returned as 1.0.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (global)

INTEGER.

= 0: successful exit

< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?hentrd

Reduces a complex Hermitian matrix to Hermitian tridiagonal form.

Syntax

```
call pchentrd (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, rwork, lrwork, info )
call pzhentrd (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, rwork, lrwork, info )
```

Description

p?hentrd is a prototype version of p?hetrd which uses tailored codes (either the serial, ?hetrd, or the parallel code, p?hettrd) when adequate workspace is provided.

p?hentrd reduces a complex Hermitian matrix sub(*A*) to Hermitian tridiagonal form *T* by an unitary similarity transformation:

Q' * sub(*A*) * *Q* = *T*, where sub(*A*) = *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1).

p?hentrd is faster than p?hetrd on almost all matrices, particularly small ones (i.e. *n* < 500 * sqrt(P)), provided that enough workspace is available to use the tailored codes.

The tailored codes provide performance that is essentially independent of the input data layout.

The tailored codes place no restrictions on *ia*, *ja*, MB or NB. At present, *ia*, *ja*, MB and NB are restricted to those values allowed by p?hetrd to keep the interface simple (see the Application Notes section for more information about the restrictions).

Input Parameters

uplo (global)

CHARACTER.

Specifies whether the upper or lower triangular part of the Hermitian matrix sub(*A*) is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n (global)

INTEGER.

The number of rows and columns to be operated on, i.e. the order of the distributed submatrix sub(*A*). *n* >= 0.

a (local)

COMPLEX for pchentrd

DOUBLE COMPLEX for pzhentrd

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the Hermitian distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia (global)

INTEGER.

The row index in the global array *a* indicating the first row of sub(*A*).

ja (global)

INTEGER.

The column index in the global array *a* indicating the first column of sub(*A*).

desca (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

work (local)

COMPLEX for pchentrd

DOUBLE COMPLEX for pzhentrd

Array, size (*lwork*)

lwork (local or global)

INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork* >= MAX(NB * (NP +1), 3 * NB).

For optimal performance, greater workspace is needed:

lwork >= 2*(*ANB*+1)*(4**NPS*+2) + (*NPS* + 4) * *NPS*

ANB = pjlaenv(*ICTXT*, 3, 'p?hettrd', 'L', 0, 0, 0, 0)

ICTXT = *desca*(*ctxt_*)

SQNPC = INT(sqrt(REAL(*NPROW* * *NPCOL*)))

NPS = MAX(numroc(*n*, 1, 0, 0, *SQNPC*), 2**ANB*)

numroc is a ScaLAPACK tool function.

pjlaenv is a ScaLAPACK environmental inquiry function.

NPROW and *NPCOL* can be determined by calling the subroutine blacs_gridinfo.

rwork (local)

COMPLEX for pchentrd

DOUBLE COMPLEX for pzhentrd

Array, size (*lrwork*)

lrwork (local or global)

INTEGER.

The size of the array *rwork*.

lrwork is local input and must be at least *lrwork* >= 1.

For optimal performance, greater workspace is needed, i.e. *lrwork* >= MAX(2 * *n*)

Output Parameters

a On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors; if *uplo* = 'L', the diagonal and first subdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors. See Application Notes.

d (local)

REAL for pchentrd

DOUBLE PRECISION for pzhentrd

Array, size LOCc(*ja*+*n*-1)

The diagonal elements of the tridiagonal matrix *T*: *d*(*i*) = *A*(*i*,*i*). *d* is tied to the distributed matrix *A*.

e (local)

REAL for pchentrd

DOUBLE PRECISION for pzhentrd

Array, size LOCc(*ja*+*n*-1) if *uplo* = 'U', LOCc(*ja*+*n*-2) otherwise.

The off-diagonal elements of the tridiagonal matrix *T*: *e*(*i*) = *A*(*i*,*i*+1) if *uplo* = 'U', *e*(*i*) = *A*(*i*+1,*i*) if *uplo* = 'L'. *e* is tied to the distributed matrix *A*.

tau (local)

COMPLEX for pchentrd

DOUBLE COMPLEX for pzhentrd

Array, size LOCc(*ja*+*n*-1).

This array contains the scalar factors *tau* of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work	On exit, *work*(1) returns the optimal *lwork*.
rwork	On exit, *rwork*(1) returns the optimal *lrwork*.
info	(global) INTEGER. = 0: successful exit < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

Q = H(*n*-1) . . . H(2) H(1).

Each H(i) has the form

H(i) = I - *tau* * v * v', where *tau* is a complex scalar, and v is a complex vector with v(i+1:*n*) = 0 and v(i) = 1; v(1:i-1) is stored on exit in *A*(*ia*:*ia*+i-2,*ja*+i), and *tau* in *tau*(*ja*+i-1).

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

Q = H(1) H(2) . . . H(*n*-1).

Each H(i) has the form

H(i) = I - *tau* * v * v', where *tau* is a complex scalar, and v is a complex vector with v(1:i) = 0 and v(i+1) = 1; v(i+2:*n*) is stored on exit in *A*(*ia*+i+1:*ia*+*n*-1,*ja*+i-1), and *tau* in *tau*(*ja*+i-1).

The contents of sub(*A*) on exit are illustrated by the following examples with *n* = 5:

if *uplo* = 'U':

$$\begin{pmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v3 \\ & & & d & e \\ & & & & d \end{pmatrix}$$

if *uplo* = 'L':

$$\begin{pmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{pmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *vi* denotes an element of the vector defining H(*i*).

Alignment requirements

The distributed submatrix sub(*A*) must verify some alignment properties, namely the following expression should be true:

(*mb_a* = *nb_a* and *IROFFA* = *ICOFFA* and *IROFFA* = 0) with *IROFFA* = mod(*ia*-1, *mb_a*), and *ICOFFA* = mod(*ja*-1, *nb_a*).

p?hetrd

Reduces a Hermitian matrix to Hermitian tridiagonal form by a unitary similarity transformation.

Syntax

```
call pchetrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pzhetrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Include Files

Description

The `p?hetrd` routine reduces a complex Hermitian matrix sub(*A*) to Hermitian tridiagonal form *T* by a unitary similarity transformation:

Q'*sub(*A*)**Q* = *T*

where sub(*A*) = *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1).

Input Parameters

uplo — (global) `CHARACTER`.

Specifies whether the upper or lower triangular part of the Hermitian matrix sub(*A*) is stored:

If *uplo* = `'U'`, upper triangular

If *uplo* = `'L'`, lower triangular

n — (global) `INTEGER`. The order of the distributed matrix sub(*A*) (*n*≥0).

a — (local)

`COMPLEX` for `pchetrd`

`DOUBLE COMPLEX` for `pzhetrd`.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the Hermitian distributed matrix sub(*A*).

If *uplo* = `'U'`, the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If *uplo* = `'L'`, the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. (see *Application Notes* below).

ia, *ja* — (global) `INTEGER`. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca — (global and local) `INTEGER` array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work — (local)

`COMPLEX` for `pchetrd`

`DOUBLE COMPLEX` for `pzhetrd`.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

lwork≥max(NB*(*np* +1), 3*NB)

where NB = *mb_a* = *nb_a*,

np = numroc(*n*, NB, MYROW, *iarow*, NPROW),

iarow = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW).

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit,

If *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors;if *uplo* = 'L', the diagonal and first subdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

d

(local)

REAL for pchetrd

DOUBLE PRECISION for pzhetrd.

Arrays of size *LOCc*(*ja*+*n*-1). The diagonal elements of the tridiagonal matrix *T*:

d(*i*)= *A*(*i*,*i*).

d is tied to the distributed matrix *A*.

e

(local)

REAL for pchetrd

DOUBLE PRECISION for pzhetrd.

Arrays of size *LOCc*(*ja*+*n*-1) if *uplo* = 'U'; *LOCc*(*ja*+*n*-2) - otherwise.

The off-diagonal elements of the tridiagonal matrix *T*:

e(*i*)= *A*(*i*,*i*+1) if *uplo* = 'U',

e(*i*) = *A*(*i*+1,*i*) if *uplo* = 'L'.

e is tied to the distributed matrix *A*.

tau

(local) COMPLEX for pchetrd

DOUBLE COMPLEX for pzhetrd.

	Array of size `LOCc(ja+n-1)`. This array contains the scalar factors of the elementary reflectors. `tau` is tied to the distributed matrix *A*.
`work(1)`	On exit `work(1)` contains the minimum value of `lwork` required for optimum performance.
`info`	(global) `INTEGER`. `= 0`: the execution is successful. `< 0`: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then `info` = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then `info` = -*i*.

Application Notes

If `uplo = 'U'`, the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(`n`-1)*...**H*(2)**H*(1).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau***v***v*',

where *tau* is a complex scalar, and *v* is a complex vector with *v*(*i*+1:`n`) = 0 and *v*(*i*) = 1; *v*(1:*i*-1) is stored on exit in *A*(`ia`:`ia`+*i*-2, `ja`+*i*), and *tau* in `tau`(`ja`+*i*-1).

If `uplo = 'L'`, the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1)**H*(2)*...**H*(`n`-1).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau***v***v*',

where *tau* is a complex scalar, and *v* is a complex vector with *v*(1:*i*) = 0 and *v*(*i*+1) = 1; *v*(*i*+2:`n`) is stored on exit in *A*(`ia`+*i*+1:`ia`+`n`-1,`ja`+*i*-1), and *tau* in `tau`(`ja`+*i*-1).

The contents of sub(*A*) on exit are illustrated by the following examples with `n = 5`:

If `uplo = 'U'`:

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If `uplo = 'L'`:

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *vi* denotes an element of the vector defining *H*(*i*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmtr

Multiplies a general matrix by the unitary transformation matrix from a reduction to tridiagonal form determined by `p?hetrd`.

Syntax

`call pcunmtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)`

`call pzunmtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)`

Include Files

Description

This routine overwrites the general complex distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'.

Q is defined as the product of *nq*-1 elementary reflectors, as returned by `p?hetrd`.

If *uplo* = 'U', *Q* = *H*(*nq*-1)... *H*(2) *H*(1);

If *uplo* = 'L', *Q* = *H*(1) *H*(2)... *H*(*nq*-1).

Input Parameters

side (global) CHARACTER

='L': *Q* or Q^H is applied from the left.

='R': *Q* or Q^H is applied from the right.

trans (global) CHARACTER

='N', no transpose, *Q* is applied.

='C', conjugate transpose, Q^H is applied.

uplo (global) CHARACTER.

= 'U': Upper triangle of *A*(*ia*:*, *ja*:*) contains elementary reflectors from p?hetrd;

= 'L': Lower triangle of *A*(*ia*:*,*ja*:*) contains elementary reflectors from p?hetrd

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

a (local)

REAL for pcunmtr

DOUBLE PRECISION for pzunmtr.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*,*LOCc*(*ja*+*n*-1)) if *side* = 'R'.

Contains the vectors which define the elementary reflectors, as returned by p?hetrd.

If *side*='L', *lld_a* ≥ max(1,*LOCr*(*ia*+*m*-1));

If *side* ='R', *lld_a* ≥ max(1,*LOCr*(*ia*+*n*-1)).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcunmtr

DOUBLE COMPLEX for pzunmtr.

Array of size of *ltau* where

If *side* = 'L' and *uplo* = 'U', *ltau* = *LOCc*(*m_a*),

if *side* = 'L' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*m*-2),

if *side* = 'R' and *uplo* = 'U', *ltau* = *LOCc*(*n_a*),

if *side* = 'R' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*n*-2).

tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by p?hetrd. *tau* is tied to the distributed matrix *A*.

c (local) COMPLEX for pcunmtr

DOUBLE COMPLEX for pzunmtr.

Pointer into the local memory to an array of size (*lld_c*,*LOCc*(*jc*+*n*-1)). Contains the local pieces of the distributed matrix sub (*C*).

ic, *jc*

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

COMPLEX for pcunmtr

DOUBLE COMPLEX for pzunmtr.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

If *uplo* = 'U',

iaa= *ia*; *jaa*= *ja*+1, *icc*= *ic*; *jcc*= *jc*;

else *uplo* = 'L',

iaa= *ia*+1, *jaa*= *ja*;

If *side* = 'L',

icc= *ic*+1; *jcc*= *jc*;

else *icc*= *ic*; *jcc*= *jc*+1;

end if

end if

If *side* = 'L',

mi= *m*-1; *ni*= *n*

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a***nb_a*

else

If *side* = 'R',

mi= *m*; *mi* = *n*-1;

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0+numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if

where *lcmq* = *lcm*/NPCOL with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),

icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q**sub(*C*), or *Q'**sub(*C*), or sub(*C*)**Q',* or sub(*C*)**Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?stebz

Computes the eigenvalues of a symmetric tridiagonal matrix by bisection.

Syntax

call psstebz(*ictxt*, *range*, *order*, *n*, *vl*, *vu*, *il*, *iu*, *abstol*, *d*, *e*, *m*, *nsplit*, *w*, *iblock*, *isplit*, *work*, *iwork*, *liwork*, *info*)

call pdstebz(*ictxt*, *range*, *order*, *n*, *vl*, *vu*, *il*, *iu*, *abstol*, *d*, *e*, *m*, *nsplit*, *w*, *iblock*, *isplit*, *work*, *iwork*, *liwork*, *info*)

Include Files

Description

The p?stebz routine computes the eigenvalues of a symmetric tridiagonal matrix in parallel. These may be all eigenvalues, all eigenvalues in the interval [*vlvu*], or the eigenvalues indexed *il* through *iu*. A static partitioning of work is done at the beginning of p?stebz which results in all processes finding an (almost) equal number of eigenvalues.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

Input Parameters

ictxt (global) INTEGER. The BLACS context handle.

range (global) CHARACTER. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues in the interval [*vl*, *vu*].

If *range* ='I', the routine computes eigenvalues with indices *il* through *iu*.

order (global) CHARACTER. Must be 'B' or 'E'.

If *order* = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block.

If *order* = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.

n (global) INTEGER. The order of the tridiagonal matrix T ($n \geq 0$).

vl, *vu* (global)

REAL for psstebz

DOUBLE PRECISION for pdstebz.

If *range* = 'V', the routine computes the lower and the upper bounds for the eigenvalues on the interval [*1*, *vu*].

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* (global)

INTEGER. Constraint: $1 \leq il \leq iu \leq n$.

If *range* = 'I', the index of the smallest eigenvalue is returned for *il* and of the largest eigenvalue for *iu* (assuming that the eigenvalues are in ascending order) must be returned.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol (global)

REAL for psstebz

DOUBLE PRECISION for pdstebz.

The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width *abstol*. If $abstol \leq 0$, then the tolerance is taken as $ulp||T||$, where *ulp* is the machine precision, and $||T||$ means the 1-norm of T

Eigenvalues will be computed most accurately when *abstol* is set to the underflow threshold slamch('U'), not 0. Note that if eigenvectors are desired later by inverse iteration (p?stein), *abstol* should be set to 2*p?lamch('S').

d (global)

REAL for psstebz

DOUBLE PRECISION for pdstebz.

Array of size *n*.

Contains *n* diagonal elements of the tridiagonal matrix *T*. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than the $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

e (global)

REAL for psstebz

DOUBLE PRECISION for pdstebz.

Array of size *n* - 1.

Contains (*n*-1) off-diagonal elements of the tridiagonal matrix *T*. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

work (local)

REAL for psstebz

DOUBLE PRECISION for pdstebz.

Array of size max(5*n*, 7). This is a workspace array.

lwork (local) INTEGER. The size of the *work* array must be ≥ max(5*n*, 7).

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

iwork (local) INTEGER. Array of size max(4*n*, 14). This is a workspace array.

liwork (local) INTEGER. The size of the *iwork* array must ≥max(4*n*, 14, NPROCS).

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

m (global) INTEGER. The actual number of eigenvalues found. 0≤*m*≤*n*

nsplit (global) INTEGER. The number of diagonal blocks detected in *T*. $1 \le nsplit \le n$

w (global)

REAL for psstebz

DOUBLE PRECISION for pdstebz.

Array of size n. On exit, the first m elements of w contain the eigenvalues on all processes.

iblock (global) INTEGER.

Array of size n. At each row/column j where $e(j)$ is zero or small, the matrix *T* is considered to split into a block diagonal matrix. On exit $iblock(i)$ specifies which block (from 1 to the number of blocks) the eigenvalue $w(i)$ belongs to.

NOTE
In the (theoretically impossible) event that bisection does not converge for some or all eigenvalues, *info* is set to 1 and the ones for which it did not are identified by a negative block number.

isplit (global) INTEGER.

Array of size n.

Contains the splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, and so on, and the $nsplit$-th submatrix consists of rows/columns $isplit(nsplit-1)+1$ through $isplit(nsplit)=n$. (Only the first $nsplit$ elements are used, but since the $nsplit$ values are not known, n words must be reserved for *isplit*.)

info (global) INTEGER.

If $info = 0$, the execution is successful.

If $info < 0$, if $info = -i$, the i-th argument has an illegal value.

If $info > 0$, some or all of the eigenvalues fail to converge or are not computed.

If $info = 1$, bisection fails to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.

If $info = 2$, mismatch between the number of eigenvalues output and the number desired.

If $info = 3$: $range='I'$, and the Gershgorin interval initially used is incorrect. No eigenvalues are computed. Probable cause: the machine has a sloppy floating-point arithmetic. Increase the *fudge* parameter, recompile, and try again.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?stedc

Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix in parallel.

Syntax

```
call psstedc (compz, n, d, e, q, iq, jq, descq, work, lwork, iwork, liwork, info )
call pdstedc (compz, n, d, e, q, iq, jq, descq, work, lwork, iwork, liwork, info )
```

Description

`p?stedc` computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix in parallel, using the divide and conquer algorithm.

Input Parameters

compz

CHARACTER*1.

= 'N': Compute eigenvalues only. (NOT IMPLEMENTED YET)

= 'I': Compute eigenvectors of tridiagonal matrix also.

= 'V': Compute eigenvectors of original dense symmetric matrix also. On entry, *Z* contains the orthogonal matrix used to reduce the original matrix to tridiagonal form. (NOT IMPLEMENTED YET)

n

(global)

INTEGER.

The order of the tridiagonal matrix *T*. *n* >= 0.

d

(global)

`REAL` for `psstedc`

`DOUBLE PRECISION` for `pdstedc`

Array, size (*n*)

On entry, the diagonal elements of the tridiagonal matrix.

e

(global)

`REAL` for `psstedc`

`DOUBLE PRECISION` for `pdstedc`

Array, size (*n*-1).

On entry, the subdiagonal elements of the tridiagonal matrix.

iq

(global)

INTEGER.

Q's global row index, which points to the beginning of the submatrix which is to be operated on.

jq

(global)

INTEGER.

Q's global column index, which points to the beginning of the submatrix which is to be operated on.

descq

(global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *Q*.

work

(local)

`REAL` for `psstedc`

`DOUBLE PRECISION` for `pdstedc`

Array, size (*lwork*)

lwork

(local)

INTEGER.

The size of the array *work*.

lwork = 6**n* + 2*NP*NQ

NP = `numroc`(*n*, NB, MYROW, DESCQ(*rsrc_*), NPROW)

NQ = `numroc`(*n*, NB, MYCOL, DESCQ(*csrc_*), NPCOL)

`numroc` is a ScaLAPACK tool function.

If *lwork* = -1, the *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum size for the *work* array. The required workspace is returned as the first element of *work* and no error message is issued by `pxerbla`.

iwork

(local)

INTEGER.

Array, size (*liwork*)

liwork

INTEGER.

The size of the array *iwork*.

liwork = 2 + 7**n* + 8*NPCOL

Output Parameters

d

On exit, if *info* = 0, the eigenvalues in descending order.

q

(local)

`REAL` for `psstedc`

`DOUBLE PRECISION` for `pdstedc`

Array, local size (*lld_q*, LOCc(*jq*+*n*-1))

q contains the orthonormal eigenvectors of the symmetric tridiagonal matrix.

On output, *q* is distributed across the P processes in block cyclic format.

work	On output, *work*(1) returns the workspace needed.
iwork	On exit, if *liwork* > 0, *iwork*(1) returns the optimal *liwork*.
info	(global) INTEGER. = 0: successful exit. < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. > 0: The algorithm failed to compute the *info*/(*n*+1)-th eigenvalue while working on the submatrix lying in global rows and columns mod(*info*,*n*+1).

p?stein

Computes the eigenvectors of a tridiagonal matrix using inverse iteration.

Syntax

```
call psstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info)
call pdstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info)
call pcstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info)
call pzstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info)
```

Include Files

Description

The p?stein routine computes the eigenvectors of a symmetric tridiagonal matrix *T* corresponding to specified eigenvalues, by inverse iteration. p?stein does not orthogonalize vectors that are on different processes. The extent of orthogonalization is controlled by the input parameter *lwork*. Eigenvectors that are to be orthogonalized are computed by the same process. p?stein decides on the allocation of work among the processes and then calls ?stein2 (modified LAPACK routine) on each individual process. If insufficient workspace is allocated, the expected orthogonalization may not be done.

NOTE
If the eigenvectors obtained are not orthogonal, increase *lwork* and run the code again.

p = NPROW*NPCOL is the total number of processes.

Input Parameters

n	(global) INTEGER. The order of the matrix *T* (*n*≥ 0).
m	(global) INTEGER. The number of eigenvectors to be returned.

d, *e*, *w*

(global)

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays:

d of size *n* contains the diagonal elements of *T*.

e of size *n*–1 contains the off-diagonal elements of *T*.

w of size *m* contains all the eigenvalues grouped by split-off block. The eigenvalues are supplied from smallest to largest within the block. (Here the output array *w* from p?stebz with order = 'B' is expected. The array should be replicated in all processes.)

iblock

(global) INTEGER.

Array of size *n*. The submatrix indices associated with the corresponding eigenvalues in *w*: 1 for eigenvalues belonging to the first submatrix from the top, 2 for those belonging to the second submatrix, etc. (The output array *iblock* from p?stebz is expected here).

isplit

(global) INTEGER.

Array of size *n*. The splitting points at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), and so on, and the *nsplit*-th submatrix consists of rows/columns *isplit*(*nsplit*–1)+1 through *isplit*(*nsplit*)=*n*. (The output array *isplit* from p?stebz is expected here.)

orfac

(global)

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

orfac specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues within *orfac**||T|| of each other are to be orthogonalized. However, if the workspace is insufficient (see *lwork*), this tolerance may be decreased until all eigenvectors can be stored in one process. No orthogonalization is done if *orfac* is equal to zero. A default value of 1000 is used if *orfac* is negative. *orfac* should be identical on all processes

iz, *jz*

(global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*.

work

(local). REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Workspace array of size *lwork*.

lwork

(local) INTEGER.

lwork controls the extent of orthogonalization which can be done. The number of eigenvectors for which storage is allocated on each process is *nvec* = floor((*lwork*-max(5*n*,*np*00*mq*00))/*n*). Eigenvectors corresponding to eigenvalue clusters of size (*nvec* - ceil(*m*/*p*) + 1) are guaranteed to be orthogonal (the orthogonality is similar to that obtained from ?stein2).

NOTE
lwork must be no smaller than max(5*n*,*np*00*mq*00) + ceil(*m*/*p*)**n* and should have the same input value on all processes.

It is the minimum value of *lwork* input on different processes that is significant.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

iwork

(local) INTEGER.

Workspace array of size 3*n*+*p*+1.

liwork

(local) INTEGER. The size of the array *iwork*. It must be greater than 3**n* +*p*+1.

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

z

(local)

REAL for psstein

DOUBLE PRECISION for pdstein

COMPLEX for pcstein

DOUBLE COMPLEX for pzstein.

Array of size *descz*(*dlen_*), *n*/NPCOL + NB). *z* contains the computed eigenvectors associated with the specified eigenvalues. Any vector which fails to converge is set to its current iterate after MAXIT iterations (See ?stein2). On output, *z* is distributed across the *p* processes in block cyclic format.

work(1)[0]

On exit, *work*(1) gives a lower bound on the workspace (*lwork*) that guarantees the user desired orthogonalization (see *orfac*). Note that this may overestimate the minimum workspace needed.

iwork

On exit, *iwork*(1) contains the amount of integer workspace required.

On exit, the *iwork*(2) through *iwork*(*p*+2) indicate the eigenvectors computed by each process. Process *i* computes eigenvectors indexed *iwork*(*i*+2)+1 through *iwork*(*i*+3).

ifail

(global) INTEGER. Array of size *m*. On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after MAXIT iterations (as in ?stein), then *info* > 0 is returned. If mod(*info*, *m*+1)>0, then for *i*=1 to mod(*info*,*m*+1), the eigenvector corresponding to the eigenvalue *w*(*ifail*(i)) failed to converge (*w* refers to the array of eigenvalues on output).

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

iclustr

(global) INTEGER. Array of size 2**p*.

This output array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be orthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*I-1) to *iclustr*(2*I), *i* = 1 to *info*/(*m*+1), could not be orthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr* is a zero terminated array: *iclustr*(2**k*)≠ 0 and *iclustr*(2**k*+1) = 0 if and only if *k* is the number of clusters.

gap

(global)

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

This output array contains the gap between eigenvalues whose eigenvectors could not be orthogonalized. The *info*/*m* output values in this array correspond to the *info*/(*m*+1) clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i-th* cluster may be as high as (*O*(*n*)**macheps*)/*gap*(*i*).

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*),

If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0: if mod(*info*, *m*+1) = *i*, then *i* eigenvectors failed to converge in MAXIT iterations. Their indices are stored in the array *ifail*. If *info*/(*m*+1) = *i*, then eigenvectors corresponding to *i* clusters of eigenvalues could not be orthogonalized due to insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Nonsymmetric Eigenvalue Problems: ScaLAPACK Computational Routines

This section describes ScaLAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

To solve a nonsymmetric eigenvalue problem with ScaLAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained.

Table "Computational Routines for Solving Nonsymmetric Eigenproblems"lists ScaLAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$, as well as routines for solving eigenproblems with Hessenberg matrices, and multiplying the matrix after reduction.

Computational Routines for Solving Nonsymmetric Eigenproblems

Operation performed	General matrix	Orthogonal/Unitary matrix	Hessenberg matrix
Reduce to Hessenberg form $A = QHQ^H$	p?gehrd		
Multiply the matrix after reduction		p?ormhr/ p?unmhr	
Find eigenvalues and Schur factorization			p?lahqr, p?hseqr

p?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

```
call psgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pdgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pcgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pzgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The p?gehrd routine reduces a real/complex general distributed matrix sub(*A*) to upper Hessenberg form *H* by an orthogonal or unitary similarity transformation

Q'*sub(*A*)**Q* = *H*,

where sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

Input Parameters

n — (global)INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).

ilo, *ihi* — (global)INTEGER.

It is assumed that sub(*A*) is already upper triangular in rows *ia*:*ia*+*ilo*-2 and *ia*+*ihi*:*ia*+*n*-1 and columns *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-1. (See *Application Notes* below).

If *n* > 0, 1≤*ilo*≤*ihi*≤*n*; otherwise set *ilo* = 1, *ihi* = *n*.

a — (local) REAL for psgehrd

DOUBLE PRECISION for pdgehrd

COMPLEX for pcgehrd

DOUBLE COMPLEX for pzgehrd.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* general distributed matrix sub(*A*) to be reduced.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work

(local)

REAL for psgehrd

DOUBLE PRECISION for pdgehrd

COMPLEX for pcgehrd

DOUBLE COMPLEX for pzgehrd.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER, size of the array *work*. *lwork* is local input and must be at least

lwork≥NB*NB + NB*max(*ihip*+1, *ihlp*+*inlq*)

where NB = *mb_a* = *nb_a*,

iroffa = mod(*ia*-1, NB),

icoffa = mod(*ja*-1, NB),

ioff = mod(*ia*+*ilo*-2, NB), *iarow* = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW), *ihip* = numroc(*ihi*+*iroffa*, NB, MYROW, *iarow*, NPROW),

ilrow = indxg2p(*ia*+*ilo*-1, NB, MYROW, *rsrc_a*, NPROW),

ihlp = numroc(*ihi*-*ilo*+*ioff*+1, NB, MYROW, *ilrow*, NPROW),

ilcol = indxg2p(*ja*+*ilo*-1, NB, MYCOL, *csrc_a*, NPCOL),

inlq = numroc(*n*-*ilo*+*ioff*+1, NB, MYCOL, *ilcol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	On exit, the upper triangle and the first subdiagonal of sub(*A*) are overwritten with the upper Hessenberg matrix *H*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).
tau	(local). REAL for psgehrd DOUBLE PRECISION for pdgehrd COMPLEX for pcgehrd DOUBLE COMPLEX for pzgehrd. Array of size at least max(*ja*+*n*-2). The scalar factors of the elementary reflectors (see *Application Notes* below). Elements *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-2 of the global vector *tau* are set to zero. *tau* is tied to the distributed matrix *A*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of (*ihi*-*ilo*) elementary reflectors

Q = *H*(*ilo*)**H*(*ilo*+1)*...**H*(*ihi*-1).

Each *H*(*i*) has the form

H(*i*)= *i* - *tau***v***v'*

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*)= 0, *v*(*i*+1)= 1 and *v*(*ihi*+1:*n*)= 0; *v*(*i*+2:*ihi*) is stored on exit in *a*(*ia*+*ilo*+*i*:*ia*+*ihi*-1,*ja*+*ilo*+*i*-2), and *tau* in *tau*(*ja*+*ilo*+*i*-2). The contents of

a(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1) are illustrated by the following example, with *n* = 7, *ilo* = 2 and *ihi* = 6:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & & & & & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & a & h & h & h & a \\ & a & h & h & h & h & a \\ & h & h & h & h & h & h \\ & v2 & h & h & h & h & h \\ & v2 & v3 & h & h & h & h \\ & v2 & v3 & v4 & h & h & h \\ & & & & & & a \end{bmatrix}$$

where *a* denotes an element of the original matrix sub(*A*), *H* denotes a modified element of the upper Hessenberg matrix *H*, and *vi* denotes an element of the vector defining *H*(*ja*+*ilo*+*i*-2).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormhr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to Hessenberg form determined by `p?gehrd`.

Syntax

`call psormhr(`*side*`, `*trans*`, `*m*`, `*n*`, `*ilo*`, `*ihi*`, `*a*`, `*ia*`, `*ja*`, `*desca*`, `*tau*`, `*c*`, `*ic*`, `*jc*`, `*descc*`, `*work*`, `*lwork*`, `*info*`)`

`call pdormhr(`*side*`, `*trans*`, `*m*`, `*n*`, `*ilo*`, `*ihi*`, `*a*`, `*ia*`, `*ja*`, `*desca*`, `*tau*`, `*c*`, `*ic*`, `*jc*`, `*descc*`, `*work*`, `*lwork*`, `*info*`)`

Include Files

Description

The `p?ormhr` routine overwrites the general real distributed *m*-by-*n* matrix sub(*C*)= *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

side = `'L'` *side* = `'R'`

trans = 'N':	Q*sub(C)	sub(C)*Q
trans = 'T':	Q^T*sub(C)	sub(C)*Q^T

where Q is a real orthogonal distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'.

Q is defined as the product of *ihi*-*ilo* elementary reflectors, as returned by p?gehrd.

Q = H(*ilo*) H(*ilo*+1)... H(*ihi*-1).

Input Parameters

side
(global) CHARACTER
='L': Q or Q^T is applied from the left.
='R': Q or Q^T is applied from the right.

trans
(global) CHARACTER
='N', no transpose, Q is applied.
='T', transpose, Q^T is applied.

m
(global) INTEGER. The number of rows in the distributed matrix sub (C) (*m*≥0).

n
(global) INTEGER. The number of columns in he distributed matrix sub (C) (*n*≥0).

ilo, *ihi*
(global) INTEGER.
ilo and *ihi* must have the same values as in the previous call of p?gehrd. Q is equal to the unit matrix except for the distributed submatrix Q(*ia*+*ilo*:*ia*+*ihi*-1,*ja*+*ilo*:*ja*+*ihi*-1).
If *side* = 'L', 1≤*ilo*≤*ihi*≤max(1,*m*);
If *side* = 'R', 1≤*ilo*≤*ihi*≤max(1,*n*);
ilo and *ihi* are relative indexes.

a
(local)
REAL for psormhr
DOUBLE PRECISION for pdormhr
Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*,*LOCc*(*ja*+*n*-1)) if *side* = 'R'.
Contains the vectors which define the elementary reflectors, as returned by p?gehrd.

ia, *ja*
(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.

desca
(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

tau
(local)
REAL for psormhr

DOUBLE PRECISION for pdormhr

Array of size *LOCc*(*ja*+*m*-2) if *side* = 'L', and *LOCc*(*ja*+*n*-2) if *side* = 'R'.

tau(*j*) contains the scalar factor of the elementary reflector *H*(*j*) as returned by p?gehrd. *tau* is tied to the distributed matrix *A*.

c

(local)

REAL for psormhr

DOUBLE PRECISION for pdormhr

Pointer into the local memory to an array of size (*lld_c*,*LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*).

ic, *jc*

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

REAL for psormhr

DOUBLE PRECISION for pdormhr

Workspace array with size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*.

lwork must be at least *iaa* = *ia* + *ilo*; *jaa* = *ja*+*ilo*-1;

If *side* = 'L',

mi = *ihi*-*ilo*; *ni* = *n*; *icc* = *ic* + *ilo*; *jcc* = *jc*; *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+*mpc*0)**nb_a*) + *nb_a***nb_a*

else if *side* = 'R',

mi = *m*; *ni* = *ihi*-*ilo*; *icc* = *ic*; *jcc* = *jc* + *ilo*; *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+max(*npa*0+numroc(numroc(*ni* +*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if

where *lcmq* = *lcm*/NPCOL with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*), *icoffc* = mod(*jcc*-1, *nb_c*),

icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — sub(*C*) is overwritten by *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmhr

Multiplies a general matrix by the unitary transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

call pcunmhr(*side*, *trans*, *m*, *n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmhr(*side*, *trans*, *m*, *n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Include Files

Description

This routine overwrites the general complex distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

side ='L' *side* ='R'

trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'H':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'.

Q is defined as the product of *ihi*-*ilo* elementary reflectors, as returned by p?gehrd.

Q = *H*(*ilo*) *H*(*ilo*+1)... *H*(*ihi*-1).

Input Parameters

side

(global) CHARACTER

='L': *Q* or Q^H is applied from the left.

='R': *Q* or Q^H is applied from the right.

trans

(global) CHARACTER

='N', no transpose, *Q* is applied.

='C', conjugate transpose, Q^H is applied.

m

(global) INTEGER. The number of rows in the distributed matrix sub (*C*) (*m*≥0).

n

(global) INTEGER. The number of columns in the distributed matrix sub (*C*) (*n*≥0).

ilo, *ihi*

(global) INTEGER

These must be the same parameters *ilo* and *ihi*, respectively, as supplied to p?gehrd. *Q* is equal to the unit matrix except in the distributed submatrix*Q*(*ia*+*ilo*:*ia*+*ihi*-1,*ja*+*ilo*:*ja*+*ihi*-1).

If *side* ='L', then 1≤*ilo*≤*ihi*≤max(1,*m*).

If *side* = 'R', then 1≤*ilo*≤*ihi*≤max(1,*n*)

ilo and *ihi* are relative indexes.

a

(local)

COMPLEX for pcunmhr

DOUBLE COMPLEX for pzunmhr.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*,*LOCc*(*ja*+*n*-1)) if *side* = 'R'.

Contains the vectors which define the elementary reflectors, as returned by p?gehrd.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau

(local)

COMPLEX for pcunmhr

DOUBLE COMPLEX for pzunmhr.

Array of size *LOCc*(*ja*+*m*-2), if *side* = 'L', and *LOCc*(*ja*+*n*-2) if *side* = 'R'.

tau(*j*) contains the scalar factor of the elementary reflector *H*(*j*) as returned by p?gehrd. *tau* is tied to the distributed matrix *A*.

c (local)

COMPLEX for pcunmhr

DOUBLE COMPLEX for pzunmhr.

Pointer into the local memory to an array of size (*lld_c*,*LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*).

ic, *jc* (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for pcunmhr

DOUBLE COMPLEX for pzunmhr.

Workspace array with size *lwork*.

lwork (local or global)

The size of the array *work*.

lwork must be at least *iaa* = *ia* + *ilo*;*jaa* = *ja*+*ilo*-1;

If *side* = 'L', *mi* = *ihi*-*ilo*; *ni* = *n*; *icc* = *ic* + *ilo*; *jcc* = *jc*; *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+*mpc*0)**nb_a*) + *nb_a***nb_a*

else if *side* = 'R',

mi = *m*; *ni* = *ihi*-*ilo*; *icc* = *ic*; *jcc* = *jc* + *ilo*; *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0+numroc(numroc(*ni* +*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if

where *lcmq* = *lcm*/NPCOL with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),

icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — *C* is overwritten by *Q** sub(*C*) or *Q*'*sub(*C*) or sub(*C*)**Q*' or sub(*C*)**Q*.

work(1)) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lahqr

Computes the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form.

Syntax

call pslahqr(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *a*, *desca*, *wr*, *wi*, *iloz*, *ihiz*, *z*, *descz*, *work*, *lwork*, *iwork*, *ilwork*, *info*)

call pdlahqr(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *a*, *desca*, *wr*, *wi*, *iloz*, *ihiz*, *z*, *descz*, *work*, *lwork*, *iwork*, *ilwork*, *info*)

call pclahqr(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *a*, *desca*, *w*, *iloz*, *ihiz*, *z*, *descz*, *work*, *lwork*, *iwork*, *ilwork*, *info*)

call pzlahqr(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *a*, *desca*, *w*, *iloz*, *ihiz*, *z*, *descz*, *work*, *lwork*, *iwork*, *ilwork*, *info*)

Include Files

Description

This is an auxiliary routine used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns *ilo* and *ihi*.

NOTE
These restrictions apply to the use of p?lahqr:

- The code requires the distributed block size to be square and at least 6.
- The code requires A and Z to be distributed identically and have identical contexts.
- The matrix A must be in upper Hessenberg form. If elements below the subdiagonal are non-zero, the resulting transformations can be nonsimilar.
- All eigenvalues are distributed to all the nodes.

Input Parameters

wantt (global) LOGICAL

If *wantt*= .TRUE., the full Schur form *T* is required;

If *wantt* = .FALSE., only eigenvalues are required.

wantz (global) LOGICAL.

If *wantz*= .TRUE., the matrix of Schur vectors *Z* is required;

If *wantz* = .FALSE., Schur vectors are not required.

n (global) INTEGER. The order of the Hessenberg matrix *A* (and *z* if *wantz*). $n \geq 0$.

ilo, *ihi* (global) INTEGER.

It is assumed that *A* is already upper quasi-triangular in rows and columns *ihi*+1:*n*, and that *A*(*ilo*, *ilo*-1) = 0 (unless *ilo* = 1). p?lahqr works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *H* if *wantt* is .TRUE.. $1 \leq ilo \leq \max(1, ihi)$; $ihi \leq n$.

a (global)

REAL for pslahqr

DOUBLE PRECISION for pdlahqr

COMPLEX for pclahqr

COMPLEX*16 for pzlahqr

Array, of size (*lld*_a,*). On entry, the upper Hessenberg matrix *A*.

desca (global and local) INTEGER array of size *dlen*_. The array descriptor for the distributed matrix *A*.

iloz, *ihiz* (global) INTEGER. Specify the rows of the matrix *Z* to which transformations must be applied if *wantz* is .TRUE.. $1 \leq iloz \leq ilo$; $ihi \leq ihiz \leq n$.

z (global)REAL for pslahqr

DOUBLE PRECISION for pdlahqr

COMPLEX for pclahqr

COMPLEX*16 for pzlahqr

Array. If *wantz* is .TRUE., on entry *z* must contain the current matrix *Z* of transformations accumulated by pdhseqr. If *wantz* is .FALSE., *z* is not referenced.

descz (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix Z.

work (local)

REAL for pslahqr

DOUBLE PRECISION for pdlahqr

COMPLEX for pclahqr

COMPLEX*16 for pzlahqr

Workspace array with size *lwork*.

lwork (local) INTEGER. The size of *work*. *lwork* is assumed big enough so that *lwork*≥3*n + max(2*max(*lld_z*,*lld_a*) + 2**LOCq*(*n*), 7*ceil(*n*/*hbl*)/*lcm*(NPROW,NPCOL))).

If *lwork* = −1, then *work*(1) gets set to the above number and the code returns immediately.

iwork (global and local) INTEGER array of size *ilwork*. Not referenced and can be NULL pointer.

ilwork (local) INTEGER This holds some of the *iblk* integer arrays. Not referenced and can be NULL pointer.

Output Parameters

a On exit, if *wantt* is .TRUE., *A* is upper quasi-triangular in rows and columns *ilo*:*ihi*, with any 2-by-2 or larger diagonal blocks not yet in standard form. If *wantt* is .FALSE., the contents of *A* are unspecified on exit.

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

wr, *wi* (global replicated output)

REAL for pslahqr

DOUBLE PRECISION for pdlahqr

Arrays of size *n* each. The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi*are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i* +1)-th, with *wi*(*i*)> 0 and *wi*(*i*+1) < 0. If *wantt* is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *A*. *A* may be returned with larger diagonal blocks until the next release.

w (global replicated output)

COMPLEX for pclahqr

COMPLEX*16 for pzlahqr

	Array of size *n*. The computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *w*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *w*, say the *i*-th and (*i*+1)-th, with *w*(*i*)> 0 and *w*(*i*+1) < 0. If *wantt* is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *A*. *A* may be returned with larger diagonal blocks until the next release.
z	On exit *z* has been updated; transformations are applied only to the submatrix *Z*(*iloz*:*ihiz*, *ilo*:*ihi*).
info	(global) INTEGER. = 0: the execution is successful. < 0: the parameter number - *info* is incorrect or inconsistent > 0: p?lahqr failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30*(*ihi*–*ilo*+1) iterations; if *info* = *i*, elements *i*+1: *ihi* of *wr* and *wi* contain the eigenvalues that have been successfully computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?hseqr

Computes eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

```
call pshseqr( job, compz, n, ilo, ihi, h, desch, wr, wi, z, descz, work, lwork, iwork, liwork, info )
```

```
call pdhseqr( job, compz, n, ilo, ihi, h, desch, wr, wi, z, descz, work, lwork, iwork, liwork, info )
```

Include Files

Description

p?hseqr computes the eigenvalues of an upper Hessenberg matrix *H* and, optionally, the matrices *T* and *Z* from the Schur decomposition $H = Z*T*Z^T$, where *T* is an upper quasi-triangular matrix (the Schur form), and *Z* is the orthogonal matrix of Schur vectors.

Optionally *Z* may be postmultiplied into an input orthogonal matrix *Q* so that this routine can give the Schur factorization of a matrix *A* which has been reduced to the Hessenberg form *H* by the orthogonal matrix *Q*: $A = Q*H*Q^T = (QZ)*T*(QZ)^T$.

NOTE

These restrictions apply to the use of p?hseqr:

- The code requires the distributed block size to be square and at least 6.
- The code requires A and Z to be distributed identically and have identical contexts.
- The matrix A must be in upper Hessenberg form. If elements below the subdiagonal are non-zero, the resulting transformations can be nonsimilar.
- All eigenvalues are distributed to all the nodes.

Input Parameters

job

(global) CHARACTER*1

= 'E': compute eigenvalues only;

= 'S': compute eigenvalues and the Schur form T.

compz

(global) CHARACTER*1

= 'N': no Schur vectors are computed;

= 'I': *z* is initialized to the unit matrix and the matrix *Z* of Schur vectors of *H* is returned;

= 'V': *z* must contain an orthogonal matrix *Q* on entry, and the product *Q***Z* is returned.

n

(global) INTEGER

The order of the Hessenberg matrix *H*. $n \geq 0$.

ilo, *ihi*

(global) INTEGER

It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*. *ilo* and *ihi* are normally set by a previous call to p gebal, and then passed to p?gehrd when the matrix output by p?gebal is reduced to Hessenberg form. Otherwise *ilo* and *ihi* should be set to 1 and *n* respectively. If $n>0$, then $1 \leq ilo \leq ihi \leq n$.

If $n = 0$, then *ilo* = 1 and *ihi* = 0.

h

REAL for pshseqr

DOUBLE PRECISION for pdhseqr

(global) array of size (*desch*(*lld_*),$LOC_c(n)$)

The upper Hessenberg matrix *H*.

desch

(global and local) INTEGER array of size *dlen_*

The array descriptor for the distributed matrix *H*.

z

REAL for pshseqr

DOUBLE PRECISION for pdhseqr

(global) array

If *compz* = 'V', on entry *z* must contain the current matrix *Z* of accumulated transformations from, for example, p?gehrd.

If *compz* = 'I', on entry *z* need not be set.

descz

(global and local) INTEGER array of size *dlen_*

The array descriptor for the distributed matrix *z*.

work

REAL for pshseqr

DOUBLE PRECISION for pdhseqr

(local workspace) array of size *lwork*.

lwork

(local) INTEGER

The length of the workspace array *work*.

If *lwork* = -1, then *work*(1) gets set to the above number and the code returns immediately.

iwork (local workspace) INTEGER array of size *liwork*

liwork (local) INTEGER

The length of the workspace array *iwork*.

If *liwork*=-1, then *iwork*(1) is set to -1 and the code returns immediately.

OUTPUT Parameters

h If *job* = 'S', *H* is upper quasi-triangular in rows and columns *ilo*:*ihi*, with 1-by-1 and 2-by-2 blocks on the main diagonal. The 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with *h*(*i*,*i*) = *h*(*i*+1,*i*+1) and *h*(*i*+1,*i*)**h*(*i*,*i*+1)<0. If *info* = 0 and *job* = 'E', the contents of *h* are unspecified on exit.

wr, *wi* REAL for pshseqr

DOUBLE PRECISION for pdhseqr

(global) array of size *n*

The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)th, with *wi*(*i*)> 0 and *wi*(*i*+1) < 0. If *job* = 'S', the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*.

z REAL for pshseqr

DOUBLE PRECISION for pdhseqr

(global) array

z is updated; transformations are applied only to the submatrix *z*(*ilo*:*ihi*,*ilo*:*ihi*).

If *compz* = 'N', *z* is not referenced.

If *compz* = 'I' and *info* = 0, *z* contains the orthogonal matrix *Z* of the Schur vectors of *H*.

info INTEGER

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument had an illegal value (see also below for -7777 and -8888).

> 0: if *info* = *i*, p?hseqr failed to compute all of the eigenvalues. Elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* contain those eigenvalues which have been successfully computed. (Failures are rare.)

If *info* > 0 and *job* = 'E', then on exit, the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final, output value of *H*.

If *info* > 0 and *job* = 'S', then on exit (*) (initial value of *H*)*U = *U*(final value of *H*) where *U* is an orthogonal matrix. The final value of *H* is upper Hessenberg and quasi-triangular in rows and columns *info*+1 through *ihi*.

If *info* > 0 and *compz* = 'V', then on exit (final value of *Z*) = (initial value of *Z*)*U where *U* is the orthogonal matrix in (*) (regardless of the value of *job*.)

If *info* > 0 and *compz* = 'I', then on exit (final value of *Z*) = *U* where *U* is the orthogonal matrix in (*) (regardless of the value of *job*.)

If *info* > 0 and *compz* = 'N', then *z* is not accessed.

= -7777: p laqr0 failed to converge and p laqr1 was called instead.

= -8888: p?laqr1 failed to converge and p?laqr0 was called instead.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trevc

Computes right and/or left eigenvectors of a complex upper triangular matrix in parallel.

Syntax

```
call pctrevc (side, howmny, select, n, t, desct, vl, descvl, vr, descvr, mm, m, work, rwork, info )
call pztrevc (side, howmny, select, n, t, desct, vl, descvl, vr, descvr, mm, m, work, rwork, info )
call pdtrevc (side, howmny, select, n, t, desct, vl, descvl, vr, descvr, mm, m, work, info )
call pstrevc (side, howmny, select, n, t, desct, vl, descvl, vr, descvr, mm, m, work, info )
```

Description

p?trevc computes some or all of the right and/or left eigenvectors of a complex upper triangular matrix *T* in parallel.

The right eigenvector x and the left eigenvector y of *T* corresponding to an eigenvalue w are defined by:

*T*x = w*x,

y'**T* = w*y'

where y' denotes the conjugate transpose of the vector y.

If all eigenvectors are requested, the routine may either return the matrices *X* and/or *Y* of right or left eigenvectors of *T*, or the products *Q*X* and/or *Q*Y*, where *Q* is an input unitary matrix. If *T* was obtained from the Schur factorization of an original matrix *A* = *Q*T*Q', then *Q*X* and *Q*Y* are the matrices of right or left eigenvectors of *A*.

Input Parameters

side (global)

CHARACTER*1.

= 'R': compute right eigenvectors only;

= 'L': compute left eigenvectors only;

= 'B': compute both right and left eigenvectors.

howmny (global)

CHARACTER*1.

= 'A': compute all right and/or left eigenvectors;

= 'B': compute all right and/or left eigenvectors, and backtransform them using the input matrices supplied in *vr* and/or *vl*;

= 'S': compute selected right and/or left eigenvectors, specified by the logical array *select*.

select (global)

LOGICAL.

Array, size (*n*)

If *howmny* = 'S', *select* specifies the eigenvectors to be computed.

If *howmny* = 'A' or 'B', *select* is not referenced. To select the eigenvector corresponding to the *j*-th eigenvalue, *select*(*j*) must be set to .TRUE..

n (global)

INTEGER.

The order of the matrix *T*. *n* >= 0.

t (local)

COMPLEX for pctrevc

DOUBLE COMPLEX for pztrevc

DOUBLE PRECISION for pdtrevc

REAL for pstrevc

Array, size (*lld_t*,*LOCc*(*n*)).

The upper triangular matrix *T*. *T* is modified, but restored on exit.

desct (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *T*.

vl (local)

COMPLEX for pctrevc

DOUBLE COMPLEX for pztrevc

DOUBLE PRECISION for pdtrevc

REAL for pstrevc

Array, size (*descvl*(*lld_*),*mm*)

On entry, if *side* = 'L' or 'B' and *howmny* = 'B', *vl* must contain an *n*-by-*n* matrix *Q* (usually the unitary matrix *Q* of Schur vectors returned by ?hseqr).

descvl

(global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *VL*.

vr

(local)

COMPLEX for pctrevc

DOUBLE COMPLEX for pztrevc

DOUBLE PRECISION for pdtrevc

REAL for pstrevc

Array, size (*descvr*(*lld_*),*mm*).

On entry, if *side* = 'R' or 'B' and *howmny* = 'B', *vr* must contain an *n*-by-*n* matrix *Q* (usually the unitary matrix *Q* of Schur vectors returned by ?hseqr).

descvr

(global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *VR*.

mm

(global)

INTEGER.

The number of columns in the arrays *vl* and/or *vr*. *mm* >= *m*.

work

(local)

COMPLEX for pctrevc

DOUBLE COMPLEX for pztrevc

DOUBLE PRECISION for pdtrevc

REAL for pstrevc

Array, size (2**desct*(*lld_*))

Additional workspace may be required if p?lattrs is updated to use *work*.

rwork

REAL for pctrevc

DOUBLE PRECISION for pztrevc

Array, size (*desct*(*lld_*))

Output Parameters

t

The upper triangular matrix *T*. *T* is modified, but restored on exit.

vl

On exit, if *side* = 'L' or 'B', *vl* contains:

if *howmny* = 'A', the matrix *Y* of left eigenvectors of *T*;

if *howmny* = 'B', the matrix *Q***Y*;

if *howmny* = 'S', the left eigenvectors of *T* specified by *select*, stored consecutively in the columns of *vl*, in the same order as their eigenvalues. If *side* = 'R', *vl* is not referenced.

vr

On exit, if *side* = 'R' or 'B', *vr* contains:

if *howmny* = 'A', the matrix *X* of right eigenvectors of *T*;

if *howmny* = 'B', the matrix *Q***X*;

if *howmny* = 'S', the right eigenvectors of *T* specified by *select*, stored consecutively in the columns of *vr*, in the same order as their eigenvalues. If *side* = 'L', *vr* is not referenced.

m

(global)

INTEGER.

The number of columns in the arrays *vl* and/or *vr* actually used to store the eigenvectors. If *howmny* = 'A' or 'B', *m* is set to *n*. Each selected eigenvector occupies one column.

info

(global)

INTEGER.

= 0: successful exit

< 0: if *info* = -i, the i-th argument had an illegal value

Application Notes

The algorithm used in this program is basically backward (forward) substitution. Scaling should be used to make the code robust against possible overflow. But scaling has not yet been implemented in p?lattrs which is called by this routine to solve the triangular systems. p?lattrs just calls p?trsv.

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be |x| + |y|.

Singular Value Decomposition: ScaLAPACK Driver Routines

This section describes ScaLAPACK routines for computing the singular value decomposition (SVD) of a general *m*-by-*n* matrix *A* (see LAPACK"Singular Value Decomposition").

To find the SVD of a general matrix *A*, this matrix is first reduced to a bidiagonal matrix *B* by a unitary (orthogonal) transformation, and then SVD of the bidiagonal matrix is computed. Note that the SVD of *B* is computed using the LAPACK routine ?bdsqr .

Table "Computational Routines for Singular Value Decomposition (SVD)" lists ScaLAPACK computational routines for performing this decomposition.

Computational Routines for Singular Value Decomposition (SVD)

Operation	General matrix	Orthogonal/unitary matrix
Reduce *A* to a bidiagonal matrix	p?gebrd	
Multiply matrix after reduction		p?ormbr/p?unmbr

p?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

```
call psgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pdgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pcgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pzgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

Include Files

Description

The p?gebrd routine reduces a real/complex general *m*-by-*n* distributed matrix sub(*A*)= *A*(*ia*:*ia*+*m*-1, *ja*:*ja* +*n*-1) to upper or lower bidiagonal form *B* by an orthogonal/unitary transformation:

Q'*sub(*A*)**P* = *B*.

If *m*≥ *n*, *B* is upper bidiagonal; if *m* < *n*, *B* is lower bidiagonal.

Input Parameters

m

(global) INTEGER. The number of rows in the distributed matrix sub(*A*) (*m*≥0).

n

(global) INTEGER. The number of columns in the distributed matrix sub(*A*) (*n*≥0).

a

(local)

REAL for psgebrd

DOUBLE PRECISION for pdgebrd

COMPLEX for pcgebrd

DOUBLE COMPLEX for pzgebrd.

Real pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja* +*n*-1)). On entry, this array contains the distributed matrix sub (*A*).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work

(local)

REAL for psgebrd

DOUBLE PRECISION for pdgebrd

COMPLEX for pcgebrd

DOUBLE COMPLEX for pzgebrd.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

lwork≥*nb**(*mpa*0 + nqa0+1)+ *nqa*0

where *nb* = *mb_a* = *nb_a*,

```
iroffa = mod(ia-1, nb),
icoffa = mod(ja-1, nb),
iarow = indxg2p(ia, nb, MYROW, rsrc_a, NPROW),
iacol = indxg2p (ja, nb, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m +iroffa, nb, MYROW, iarow, NPROW),
nqa0 = numroc(n +icoffa, nb, MYCOL, iacol, NPCOL),
```

NOTE
`mod(x,y)` is the integer remainder of x/y.

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, if $m \geq n$, the diagonal and the first superdiagonal of sub(*A*) are overwritten with the upper bidiagonal matrix *B*; the elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the first superdiagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix *B*; the elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the diagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors. See *Application Notes* below.

d

(local)

`REAL` for single-precision flavors

`DOUBLE PRECISION` for double-precision flavors.

Array of size *LOCc*(*ja*+min(*m*,*n*)-1) if $m \geq n$ and *LOCr*(*ia*+min(*m*,*n*)-1) otherwise. The distributed diagonal elements of the bidiagonal matrix *B*: *d*(*i*) = *a*(*i*,*i*).

d is tied to the distributed matrix *A*.

e

(local)

`REAL` for single-precision flavors

`DOUBLE PRECISION` for double-precision flavors.

Array of size *LOCr*(*ia*+min(*m*,*n*)-1) if $m \geq n$; *LOCc*(*ja*+min(*m*,*n*)-2) otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix *B*:

	If `m≥n`, `e(i) = a(i,i+1)` for `i = 1,2,..., n-1`; if `m < n`, `e(i) = a(i+1,i)` for `i = 1,2,..., m-1`. `e` is tied to the distributed matrix *A*.

tauq, *taup*	(local)
	`REAL` for `psgebrd`
	`DOUBLE PRECISION` for `pdgebrd`
	`COMPLEX` for `pcgebrd`
	`DOUBLE COMPLEX` for `pzgebrd`.
	Arrays of size *LOCc*(*ja*+min(*m*,*n*)-1) for *tauq* and *LOCr*(*ia*+min(*m*,*n*)-1) for *taup*. Contain the scalar factors of the elementary reflectors that represent the orthogonal/unitary matrices *Q* and *P*, respectively. *tauq* and *taup* are tied to the distributed matrix *A*. See *Application Notes* below.

work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) `INTEGER`. `= 0`: the execution is successful. `< 0`: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

If `m≥n`,

Q = *H*(1)**H*(2)*...**H*(*n*), and *P* = *G*(1)**G*(2)*...**G*(*n*-1).

Each *H*(*i*) and *G*(*i*) has the form:

H(*i*)= *i* - *tauq* * *v* * *v'* and *G*(*i*) = *i* - *taup***u***u'*

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors;

v(1:*i*-1) = 0, *v*(*i*) = 1, and *v*(*i*+1:*m*) is stored on exit in *A*(*ia*+*i*:*ia*+*m*-1,*ja*+*i*-1);

u(1:*i*) = 0, *u*(*i*+1) = 1, and *u*(*i*+2:*n*) is stored on exit in *A* (*ia*+*i*-1,*ja*+*i*+1:*ja*+*n*-1);

tauq is stored in *tauq*(*ja*+*i*-1) and *taup* in *taup*(*ia*+*i*-1).

If *m* < *n*,

Q = *H*(1)**H*(2)*...**H*(*m*-1), and *P* = *G*(1)* *G*(2)*...* *G*(*m*)

Each *H* (*i*) and *G*(*i*) has the form:

H(*i*)= *i*-*tauq***v***v'* and *G*(*i*)= *i*-*taup***u***u'*

here *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors;

v(1:*i*) = 0, *v*(*i*+1) = 1, and *v*(*i*+2:*m*) is stored on exit in *A* (*ia*+*i*:*ia*+*m*-1,*ja*+*i*-1); *u*(1:*i*-1) = 0, *u*(*i*) = 1, and *u*(*i*+1:*n*) is stored on exit in *A*(*ia*+*i*-1,*ja*+*i*+1:*ja*+*n*-1);

tauq is stored in *tauq*(*ja*+*i*-1) and *taup* in *taup*(*ia*+*i*-1).

The contents of sub(*A*) on exit are illustrated by the following examples:

$m = 6$ and $n = 5$ ($m > n$):

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6$ ($m < n$):

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *B*, *vi* denotes an element of the vector defining *H*(*i*), and *ui* an element of the vector defining *G*(*i*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormbr

Multiplies a general matrix by one of the orthogonal matrices from a reduction to bidiagonal form determined by `p?gebrd`.

Syntax

```
call psormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

```
call pdormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

Include Files

Description

If *vect* = 'Q', the `p?ormbr` routine overwrites the general real distributed m-by-n matrix sub(*C*) = *C*(ic:ic +m-1,jc:jc+n-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q* sub(*C*)	sub(*C*) *Q*
trans = 'T':	Q^T sub(*C*)	sub(*C*) Q^T

If $vect$ = 'P', the routine overwrites sub(*C*) with

	side ='L'	*side* ='R'
trans = 'N':	*P* sub(*C*)	sub(*C*) *P*
trans = 'T':	P^T sub(*C*)	sub(*C*) P^T

Here *Q* and P^T are the orthogonal distributed matrices determined by p?gebrd when reducing a real distributed matrix *A*(*ia*:*, *ja*:*) to bidiagonal form: *A*(*ia*:*, *ja*:*) = *Q***B***P*^{T}. *Q* and P^T are defined as products of elementary reflectors *H*(*i*) and *G*(*i*) respectively.

Let *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'. Therefore *nq* is the order of the orthogonal matrix *Q* or P^T that is applied.

If *vect* = 'Q', *A*(*ia*:*, *ja*:*) is assumed to have been an *nq*-by-*k* matrix:

If *nq* ≥ *k*, *Q* = *H*(1) *H*(2)...*H*(*k*);

If *nq* < *k*, *Q* = *H*(1) *H*(2)...*H*(*nq*-1).

If *vect* = 'P', *A*(*ia*:*, *ja*:*) is assumed to have been a *k*-by-*nq* matrix:

If *k* < *nq*, *P* = *G*(1) *G*(2)...*G*(*k*);

If *k* ≥ *nq*, *P* = *G*(1) *G*(2)...*G*(*nq*-1).

Input Parameters

vect
(global) CHARACTER.
If *vect* ='Q', then *Q* or Q^T is applied.
If *vect* ='P', then *P* or P^T is applied.

side
(global) CHARACTER.
If *side* ='L', then *Q* or Q^T, *P* or P^T is applied from the left.
If *side* ='R', then *Q* or Q^T, *P* or P^T is applied from the right.

trans
(global) CHARACTER.
If *trans* = 'N', no transpose, *Q* or *P* is applied.
If *trans* = 'T', then Q^T or P^T is applied.

m
(global) INTEGER. The number of rows in the distributed matrix sub (*C*).

n
(global) INTEGER. The number of columns in the distributed matrix sub (*C*).

k
(global) INTEGER.
If *vect* = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd;
If *vect* = 'P', the number of rows in the original distributed matrix reduced by p?gebrd.
Constraints: *k*≥ 0.

a
(local)
REAL for psormbr
DOUBLE PRECISION for pdormbr.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+min(*nq*,*k*)-1)) if *vect*='Q', and (*lld_a*,*LOCc*(*ja*+*nq*-1)) if *vect* = 'P'.

nq = *m* if *side* = 'L', and *nq* = *n* otherwise.

The vectors that define the elementary reflectors *H*(*i*) and *G*(*i*), whose products determine the matrices *Q* and *P*, as returned by p?gebrd.

If *vect* = 'Q', *lld_a*≥max(1, *LOCr*(*ia*+*nq*-1));

If *vect* = 'P', *lld_a*≥max(1, *LOCr*(*ia*+min(*nq*, *k*)-1)).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau

(local)

REAL for psormbr

DOUBLE PRECISION for pdormbr.

Array of size *LOCc*(*ja*+min(*nq*, *k*)-1), if *vect* = 'Q', and *LOCr*(*ia*+min(*nq*, *k*)-1), if *vect* = 'P'.

tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*) or *G*(*i*)

which determines *Q* or *P*, as returned by pdgebrd in its array argument *tauq* or *taup*. *tau* is tied to the distributed matrix *A*.

c

(local) REAL for psormbr

DOUBLE PRECISION for pdormbr

Pointer into the local memory to an array of size (*lld_c*,*LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub (*C*).

ic, *jc*

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

REAL for psormbr

DOUBLE PRECISION for pdormbr.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

If *side* = 'L'

nq = *m*;

if ((*vect* = 'Q' and *nq*≥*k*) or (*vect* is not equal to 'Q' and *nq*>*k*)), *iaa*=*ia*; *jaa*=*ja*; *mi*=*m*; *ni*=*n*; *icc*=*ic*; *jcc*=*jc*;

else

iaa= *ia*+1; *jaa*=*ja*; *mi*=*m*-1; *ni*=*n*; *icc*=*ic*+1; *jcc*= *jc*;

end if

else

If *side* = 'R', *nq* = *n*;

if((*vect* = 'Q' and *nq*≥*k*) or (*vect* is not equal to 'Q' and *nq*>*k*)),

iaa=*ia*; *jaa*=*ja*; *mi*=*m*; *ni*=*n*; *icc*=*ic*; *jcc*=*jc*;

else

iaa= *ia*; *jaa*= *ja*+1; *mi*= *m*; *ni*= *n*-1; *icc*= *ic*; *jcc*= *jc*+1;

end if

end if

If *vect* = 'Q',

If *side* = 'L', *lwork*≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a* * *nb_a*

else if *side* = 'R',

lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if

else if *vect* is not equal to 'Q', if *side* = 'L',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0 + numroc(numroc(*mi*+iroffc, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* = 'R',

lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a***mb_a*

end if

end if

where *lcmp* = *lcm*/NPROW, *lcmq* = *lcm*/NPCOL, with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*jaa*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*mi*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),

icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — On exit, if *vect*='Q', sub(*C*) is overwritten by *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*; if *vect*='P', sub(*C*) is overwritten by *P**sub(*C*), or *P*'*sub(*C*), or sub(*C*)**P*, or sub(*C*)**P*'.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmbr

Multiplies a general matrix by one of the unitary transformation matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

call pcunmbr(*vect*, *side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmbr(*vect*, *side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Include Files

Description

If *vect* = 'Q', the p?unmbr routine overwrites the general complex distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

If *vect* = 'P', the routine overwrites sub(*C*) with

	side ='L'	*side* ='R'
trans = 'N':	*P**sub(*C*)	sub(*C*)**P*
trans = 'C':	P^H*sub(*C*)	sub(*C*)*P^H

Here *Q* and P^H are the unitary distributed matrices determined by p?gebrd when reducing a complex distributed matrix *A*(*ia*:*, *ja*:*) to bidiagonal form: *A*(*ia*:*, *ja*:*) = *Q***B**P^H.

Q and P^H are defined as products of elementary reflectors *H*(*i*) and *G*(*i*) respectively.

Let *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'. Therefore *nq* is the order of the unitary matrix *Q* or P^H that is applied.

If *vect* = 'Q', *A*(*ia*:*, *ja*:*) is assumed to have been an *nq*-by-*k* matrix:

If *nq* ≥ *k*, *Q* = *H*(1) *H*(2)... *H*(*k*);

If *nq* < *k*, *Q* = *H*(1) *H*(2)... *H*(*nq*-1).

If *vect* = 'P', *A*(*ia*:*, *ja*:*) is assumed to have been a *k*-by-*nq* matrix:

If *k* < *nq*, *P* = *G*(1) *G*(2)... *G*(*k*);

If *k* ≥ *nq*, *P* = *G*(1) *G*(2)... *G*(*nq*-1).

Input Parameters

vect
(global) CHARACTER.
If *vect* ='Q', then *Q* or Q^H is applied.
If *vect* ='P', then *P* or P^H is applied.

side
(global) CHARACTER.
If *side* ='L', then *Q* or Q^H, *P* or P^H is applied from the left.
If *side* ='R', then *Q* or Q^H, *P* or P^H is applied from the right.

trans
(global) CHARACTER.
If *trans* = 'N', no transpose, *Q* or *P* is applied.
If *trans* = 'C', conjugate transpose, Q^H or P^H is applied.

m
(global) INTEGER. The number of rows in the distributed matrix sub (*C*) *m*≥0.

n
(global) INTEGER. The number of columns in the distributed matrix sub (*C*) *n*≥0.

k
(global) INTEGER.
If *vect* = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd;
If *vect* = 'P', the number of rows in the original distributed matrix reduced by p?gebrd.

Constraints: $k \geq 0$.

a (local)

COMPLEX for psormbr

DOUBLE COMPLEX for pdormbr.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+min(*nq*,*k*)-1)) if *vect*='Q', and (*lld_a*,*LOCc*(*ja*+*nq*-1)) if *vect* = 'P'.

nq = *m* if *side* = 'L', and *nq* = *n* otherwise.

The vectors that define the elementary reflectors *H*(*i*) and *G*(*i*), whose products determine the matrices *Q* and *P*, as returned by p?gebrd.

If *vect* = 'Q', *lld_a* ≥ max(1, *LOCr*(*ia*+*nq*-1));

If *vect* = 'P', *lld_a* ≥ max(1, *LOCr*(*ia*+min(*nq*, *k*)-1)).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcunmbr

DOUBLE COMPLEX for pzunmbr.

Array of size *LOCc*(*ja*+min(*nq*, *k*)-1), if *vect* = 'Q', and *LOCr*(*ia*+min(*nq*, *k*)-1), if *vect* = 'P'.

tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*) or *G*(*i*), which determines *Q* or *P*, as returned by p?gebrd in its array argument *tauq* or *taup*. *tau* is tied to the distributed matrix *A*.

c (local) COMPLEX for pcunmbr

DOUBLE COMPLEX for pzunmbr

Pointer into the local memory to an array of size (*lld_c*,*LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub (*C*).

ic, *jc* (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for pcunmbr

DOUBLE COMPLEX for pzunmbr.

Workspace array of size *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least:

If *side* = 'L'

nq = *m*;

if ((*vect* = 'Q' and *nq* ≥ *k*) or (*vect* is not equal to 'Q' and *nq*>*k*)), *iaa*= *ia*; *jaa*= *ja*; *mi*= *m*; *ni*= *n*; *icc*= *ic*; *jcc*= *jc*;

else

iaa= *ia*+1; *jaa*= *ja*; *mi*= *m*-1; *ni*= *n*; *icc*= *ic*+1; *jcc*= *jc*;

end if

else

If *side* = 'R', *nq* = *n*;

if ((*vect* = 'Q' and *nq* ≥ *k*) or (*vect* is not equal to 'Q' and *nq*≥*k*)),

iaa= *ia*; *jaa*= *ja*; *mi*= *m*; *ni*= *n*; *icc*= *ic*; *jcc*= *jc*;

else

iaa= *ia*; *jaa*= *ja*+1; *mi*= *m*; *ni*= *n*-1; *icc*= *ic*; *jcc*= *jc*+1;

end if

end if

If *vect* = 'Q',

If *side* = 'L', *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+*mpc*0)**nb_a*) + *nb_a***nb_a*

else if *side* = 'R',

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0+numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if

else if *vect* is not equal to 'Q',

if *side* = 'L',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0+numroc(numroc(*mi*+iroffc, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* = 'R',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a***mb_a*

end if

end if

where *lcmp* = *lcm*/NPROW, *lcmq* = *lcm*/NPCOL, with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*jaa*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*mi*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),

icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — On exit, if *vect*='Q', sub(*C*) is overwritten by *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*; if *vect*='P', sub(*C*) is overwritten by *P**sub(*C*), or *P*'*sub(*C*), or sub(*C*)**P,* or sub(*C*)**P*'.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Generalized Symmetric-Definite Eigenvalue Problems: ScaLAPACK Computational Routines

This section describes ScaLAPACK routines that allow you to reduce the *generalized symmetric-definite eigenvalue problems* (see LAPACKGeneralized Symmetric-Definite Eigenvalue Problems) to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling ScaLAPACK routines (see Symmetric Eigenproblems).

Table "Computational Routines for Reducing Generalized Eigenproblems to Standard Problems" lists these routines.

Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to standard problems	p?sygst	p?hegst

p?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pssygst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
call pdsygst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

Include Files

Description

The p?sygstroutine reduces real symmetric-definite generalized eigenproblems to the standard form.

In the following sub(*A*) denotes *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1).

If *ibtype* = 1, the problem is

sub(*A*)*x = λ*sub(*B*)*x,

and sub(*A*) is overwritten by inv(U^T)*sub(*A*)*inv(*U*), or inv(*L*)*sub(*A*)*inv(L^T).

If *ibtype* = 2 or 3, the problem is

sub(*A*)*sub(*B*)*x = λ*x, or sub(*B*)*sub(*A*)*x = λ*x,

and sub(*A*) is overwritten by *U*sub(*A*)*U^T, or L^T*sub(*A*)**L*.

sub(*B*) must have been previously factorized as U^T**U* or *L*L^T by p?potrf.

Input Parameters

ibtype — (global) INTEGER. Must be 1 or 2 or 3.

If *itype* = 1, compute inv(U^T)*sub(*A*)*inv(*U*), or inv(*L*)*sub(*A*)*inv(L^T);

If *itype* = 2 or 3, compute *U*sub(*A*)*U^T, or L^T*sub(*A*)**L*.

uplo — (global) CHARACTER. Must be 'U' or 'L'.

If *uplo* = 'U', the upper triangle of sub(*A*) is stored and sub (*B*) is factored as U^T**U*.

If *uplo* = 'L', the lower triangle of sub(*A*) is stored and sub (*B*) is factored as *L*L^T.

n — (global) INTEGER. The order of the matrices sub (*A*) and sub (*B*) (*n*≥ 0).

a — (local)

REAL for pssygst

DOUBLE PRECISION for pdsygst.

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)). On entry, the array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

b

(local)

REAL for pssygst

DOUBLE PRECISION for pdsygst.

Pointer into the local memory to an array of size (*lld_b*,*LOCc*(*jb*+*n*-1)). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub (*B*) as returned by p?potrf.

ib, *jb*

(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

descb

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix B.

Output Parameters

a

On exit, if *info* = 0, the transformed matrix, stored in the same format as sub(*A*).

scale

(global)

REAL for pssygst

DOUBLE PRECISION for pdsygst.

Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, *scale* is always returned as 1.0, it is returned here to allow for future enhancement.

info

(global) INTEGER.

If *info* = 0, the execution is successful. If *info* < 0, if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?hegst

Reduces a Hermitian positive-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pchegst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

```
call pzhegst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

Include Files

Description

The p?hegst routine reduces complex Hermitian positive-definite generalized eigenproblems to the standard form.

In the following sub(*A*) denotes *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1).

If *ibtype* = 1, the problem is

sub(*A*)*x* = λ*sub(*B*)*x*,

and sub(*A*) is overwritten by inv(U^H)*sub(*A*)*inv(*U*), or inv(*L*)*sub(*A*)*inv(L^H).

If *ibtype* = 2 or 3, the problem is

sub(*A*)*sub(*B*)*x* = λ**x*, or sub(*B*)*sub(*A*)*x* = λ**x*,

and sub(*A*) is overwritten by *U*sub(*A*)*U^H, or L^H*sub(*A*)**L*.

sub(*B*) must have been previously factorized as U^H**U* or *L*L^H by p?potrf.

Input Parameters

ibtype

(global) INTEGER. Must be 1 or 2 or 3.

If *itype* = 1, compute inv(U^H)*sub(*A*)*inv(*U*), or inv(*L*)*sub(*A*)*inv(L^H);

If *itype* = 2 or 3, compute *U*sub(*A*)*U^H, or L^H*sub(*A*)**L*.

uplo

(global) CHARACTER. Must be 'U' or 'L'.

If *uplo* = 'U', the upper triangle of sub(*A*) is stored and sub (*B*) is factored as U^H**U*.

If *uplo* = 'L', the lower triangle of sub(*A*) is stored and sub (*B*) is factored as *L*L^H.

n

(global) INTEGER. The order of the matrices sub (*A*) and sub (*B*) (*n*≥0).

a

(local)

COMPLEX for pchegst

DOUBLE COMPLEX for pzhegst.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

b

(local)

COMPLEX for pchegst

DOUBLE COMPLEX for pzhegst.

Pointer into the local memory to an array of size (*lld_b*,*LOCc*(*jb*+*n*-1)). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub (*B*) as returned by p?potrf.

ib, *jb* — (global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

descb — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix B.

Output Parameters

a — On exit, if *info* = 0, the transformed matrix, stored in the same format as sub(*A*).

scale — (global)

REAL for pchegst

DOUBLE PRECISION for pzhegst.

Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, *scale* is always returned as 1.0, it is returned here to allow for future enhancement.

info — (global) INTEGER.

If *info* = 0, the execution is successful. If *info* <0, if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

ScaLAPACK Driver Routines

Table "ScaLAPACK Driver Routines" lists ScaLAPACK driver routines available for solving systems of linear equations, linear least-squares problems, standard eigenvalue and singular value problems, and generalized symmetric definite eigenproblems.

ScaLAPACK Driver Routines

Type of Problem	Matrix type, storage scheme	Driver
Linear equations	general (partial pivoting)	p?gesv (simple driver) / p?gesvx (expert driver)
	general band (partial pivoting)	p?gbsv (simple driver)
	general band (no pivoting)	p?dbsv (simple driver)
	general tridiagonal (no pivoting)	p?dtsv (simple driver)
	symmetric/Hermitian positive-definite	p?posv (simple driver) / p?posvx (expert driver)
	symmetric/Hermitian positive-definite, band	p?pbsv (simple driver)
	symmetric/Hermitian positive-definite, tridiagonal	p?ptsv (simple driver)
Linear least squares problem	general *m*-by-*n*	p?gels

Type of Problem	Matrix type, storage scheme	Driver
Non-symmetric eigenvalue problem	general	p?geevx (expert driver)
Symmetric eigenvalue problem	symmetric/Hermitian	p?syev / p?heev (simple driver); p?syevd / p?heevd (simple driver with a divide and conquer algorithm); p?syevx / p?heevx (expert driver); p?syevr / p?heevr (simple driver with MRRR algorithm)
Singular value decomposition	general *m*-by-*n*	p?gesvd
Generalized symmetric definite eigenvalue problem	symmetric/Hermitian, one matrix also positive-definite	p?sygvx / p?hegvx (expert driver)

p?geevx

Computes for an n-by-n real/complex non-symmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

Syntax

```
call psgeevx(balanc, jobvl, jobvr, sense, n, a, desca, wr, wi, vl, descvl, vr, descvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, info)
```

```
call pdgeevx(balanc, jobvl, jobvr, sense, n, a, desca, wr, wi, vl, descvl, vr, descvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, info)
```

```
call pcgeevx(balanc, jobvl, jobvr, sense, n, a, desca, w, vl, descvl, vr, descvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, info)
```

```
call pzgeevx(balanc, jobvl, jobvr, sense, n, a, desca, w, vl, descvl, vr, descvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, info)
```

Include Files

Description

The p?geevx routine computes for an *n*-by-*n* real/complex non-symmetric matrix *A*, the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *scale*, and *abnrm*), reciprocal condition numbers for the eigenvalues (*rconde*).

The right eigenvector *v* of *A* satisfies

$$A \cdot v = \lambda \cdot v$$

where λ is its eigenvalue.

The left eigenvector *u* of *A* satisfies.

$u^H A = \lambda u^H$

where u^H denotes the conjugate transpose of *u*. The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation *D***A**inv(*D*), where *D* is a diagonal matrix, to make its rows and columns closer in norm and the condition number of its eigenvalues smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers in exact arithmetic, but diagonal scaling will.

NOTE
The current version doesn't support computation of the reciprocal condition numbers for the right eigenvectors.

Current Notes and Restrictions

All the `p?geevx` interfaces call `p?lahqr` for computing eigenvalues and eigenvectors of the Hessenberg matrices. There are several restrictions for the usage of `p?lahqr`, which include:

- The current implementation of `p?lahqr` requires the distributed block size to be square and at least six (6); unlike simpler codes like LU, this algorithm is extremely sensitive to block size.
- The current implementation of `p?lahqr` requires that input matrix *A*, the left and right eigenvector matrices *VR* and/or *VL* to be distributed identically and have identical context.

Parameters

balanc

(global). Must be 'N', 'P', 'S', or 'B'. Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues.

If *balanc* = 'N', do not diagonally scale or permute;

If *balanc* = 'P', perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale;

If *balanc* = 'S', diagonally scale the matrix, that is, replace *A* by *D***A**`inv`(*D*), where *D* is a diagonal matrix chosen to make the rows and columns of *A* more equal in norm. Do not permute;

If *balanc* = 'B', both diagonally scale and permute *A*.

Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

jobvl

(global). Must be 'N' or 'V'.

If *jobvl* = 'N', left eigenvectors of *A* are not computed;

If *jobvl* = 'V', left eigenvectors of *A* are computed.

If *sense* = 'E', then *jobvl* must be 'V'.

jobvr

(global). Must be 'N' or 'V'.

If *jobvr* = 'N', right eigenvectors of *A* are not computed;

If *jobvr* = 'V', right eigenvectors of *A* are computed.

If *sense* = 'E', then *jobvr* must be 'V'.

sense

(global). Must be 'N' or 'E'. Determines which reciprocal condition numbers are computed.

If *sense* = 'N', none are computed.

If *sense* = 'E', computed for eigenvalues only.

n

(global) The order of the distributed matrix *A* ($n \geq 0$).

a

(local)

Pointer into the local memory to an array of size *lld_a*LOCc*(*n*). On entry, this array contains the local pieces of the *n*-by-*n* general distributed matrix *A* to be reduced.

desca (global and local) array of size dlen_. The array descriptor for the distributed matrix *A*.

wr, *wi* (global output) Arrays, size at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w (global output) Array, size at least max(1, *n*). Contains the computed eigenvalues.

vl (local output)

Pointer into the local memory to an array of size (DESCVL(LLD_),LOCc(n)).

If *jobvl* = 'N', *vl* is not referenced. If *jobvl* = 'V', the *vl* parameter contains the local pieces of the left eigenvectors of the matrix *A*.

descvl (global and local input) array of size *dlen_*. The array descriptor for the distributed matrix *vl*.

vr (local output)

Pointer into the local memory to an array of size (DESCVR(LLD_),LOCc(n)).

If *jobvr* = 'N', *vr* is not referenced. If *jobvr* = 'V', the *vr* parameter contains the local pieces of the right eigenvectors of the matrix *A*.

descvr (global and local input) array of size *dlen_*. The array descriptor for the distributed matrix *vr*.

ilo, *ihi* (global output)

ilo and *ihi* are integer values determined when *A* was balanced.

The balanced *A*(i,j) = 0 if i > j and j = 1,..., *ilo*-1 or i= *ihi*+1,..., *n*.

If *balanc* = 'N' or 'S', *ilo* = 1 and *ihi* = *n*.

scale (global output)

Array, size at least max(1, *n*). Details of the permutations and scaling factors applied when balancing *A*.

If P[*j* - 1] is the index of the row and column interchanged with row and column *j*, and D[*j* - 1] is the scaling factor applied to row and column *j*, then

scale[*j* - 1] = P[*j* - 1], for *j* = 1,...,*ilo*-1

= D[*j* - 1], for *j* = *ilo*,...,*ihi*

= P[*j* - 1] for *j* = *ihi*+1,..., *n*.

The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

abnrm The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

rconde Array, size at least max(1, *n*).

rconde[*j* - 1] is the reciprocal condition number of the *j*-th eigenvalue.

rcondv	Not supported in the current version. It could be null pointer.
work	(local) Workspace array of size *lwork*.
lwork	(local or global) size of the array *work*. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum size for the work array. These values are returned in the first entry of the *work* array, and no error message is issued by pxerbla.
info	(global) = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-th entry, indexed *j*- 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gesv

Computes the solution to the system of linear equations with a square distributed matrix and multiple right-hand sides.

Syntax

```
call psgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pzgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

Include Files

Description

The p?gesvroutine computes the solution to a real or complex system of linear equations sub(*A*)**X* = sub(*B*), where sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*) is an *n*-by-*n* distributed matrix and *X* and sub(*B*) = *B*(*ib:ib+n-1*, *jb:jb+nrhs-1*) are *n*-by-*nrhs* distributed matrices.

The *LU* decomposition with partial pivoting and row interchanges is used to factor sub(*A*) as sub(*A*) = *P***L***U*, where *P* is a permutation matrix, *L* is unit lower triangular, and *U* is upper triangular. *L* and *U* are stored in sub(*A*). The factored form of sub(*A*) is then used to solve the system of equations sub(*A*)**X* = sub(*B*).

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*) (*n*≥ 0).
nrhs	(global) INTEGER. The number of right hand sides, that is, the number of columns of the distributed submatrices *B* and *X* (*nrhs*≥ 0).
a, *b*	(local) REAL for psgesv

DOUBLE PRECISION for pdgesv

COMPLEX for pcgesv

DOUBLE COMPLEX for pzgesv.

Pointers into the local memory to arrays of local size *a*(*lld_a*, *LOCc*(*ja*+*n*-1)) and *b*(*lld_b*, *LOCc*(*jb*+*nrhs*-1)), respectively.

On entry, the array *a* contains the local pieces of the *n*-by-*n* distributed matrix sub(*A*) to be factored.

On entry, the array *b* contains the right hand side distributed matrix sub(*B*).

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of sub(*A*), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

ib, *jb* — (global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of sub(*B*), respectively.

descb — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Output Parameters

a — Overwritten by the factors *L* and *U* from the factorization sub(*A*) = *P***L***U*; the unit diagonal elements of *L* are not stored .

b — Overwritten by the solution distributed matrix *X*.

ipiv — (local) INTEGER Array of size *LOCr*(*m_a*)+*mb_a*. This array contains the pivoting information. The (local) row i of the matrix was interchanged with the (global) row *ipiv*(i).

This array is tied to the distributed matrix *A*.

info — (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info> 0:

If *info* = *k*, *U*(*ia*+*k*-1,*ja*+*k*-1) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gesvx

Uses the LU factorization to compute the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

call psgesvx(*fact*, *trans*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *ipiv*, *equed*, *r*, *c*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pdgesvx(*fact*, *trans*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *ipiv*, *equed*, *r*, *c*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pcgesvx(*fact*, *trans*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *ipiv*, *equed*, *r*, *c*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *rwork*, *lrwork*, *info*)

call pzgesvx(*fact*, *trans*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *ipiv*, *equed*, *r*, *c*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *rwork*, *lrwork*, *info*)

Include Files

Description

The p?gesvx routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations *AX = B*, where *A* denotes the *n*-by-*n* submatrix *A(ia:ia+n-1, ja:ja+n-1)*, *B* denotes the *n*-by-*nrhs* submatrix *B(ib:ib+n-1, jb:jb+nrhs-1)* and *X* denotes the *n*-by-*nrhs* submatrix *X(ix:ix+n-1, jx:jx+nrhs-1)*.

Error bounds on the solution and a condition estimate are also provided.

In the following description, *af* stands for the subarray *af(iaf:iaf+n-1, jaf:jaf+n-1)*.

The routine p?gesvx performs the following steps:

1. If *fact* = 'E', real scaling factors *R* and *C* are computed to equilibrate the system:

   trans = 'N': diag(*R*)**A**diag(*C*) *diag(*C*)-1**X* = diag(*R*)*B

   trans = 'T': (diag(*R*)**A**diag(*C*))*T* *diag(*R*)-1**X* = diag(*C*)*B

   trans = 'C': (diag(*R*)**A**diag(*C*))*H* *diag(*R*)-1**X* = diag(*C*)*B

   Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by diag(*R*)**A**diag(*C*) and *B* by diag(*R*)**B* (if *trans*='N') or diag(*c*)**B* (if *trans* = 'T' or 'C').
2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as *A* = *PLU*, where *P* is a permutation matrix, *L* is a unit lower triangular matrix, and *U* is upper triangular.
3. The factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than relative machine precision, steps 4 - 6 are skipped.
4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by diag(*C*) (if *trans* = 'N') or diag(*R*) (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact (global) CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F' then, on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*. Arrays *a*, *af*, and *ipiv* are not modified.

If *fact* = 'N', the matrix *A* is copied to *af* and factored.

If *fact* = 'E', the matrix *A* is equilibrated if necessary, then copied to *af* and factored.

trans

(global) CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form $A*X = B$ (No transpose);

If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose);

If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate transpose);

n

(global) INTEGER. The number of linear equations; the order of the submatrix *A*(*n*≥ 0).

nrhs

(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrices *B* and *X*(*nrhs*≥ 0).

a, *af*, *b*, *work*

(local)

REAL for psgesvx

DOUBLE PRECISION for pdgesvx

COMPLEX for pcgesvx

DOUBLE COMPLEX for pzgesvx.

Pointers into the local memory to arrays of local size *a*(*lld_a*,*LOCc*(*ja*+*n*-1)), *af*(*lld_af*,*LOCc*(*ja*+*n*-1)), *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-1)), *work*(*lwork*).

The array *a* contains the matrix *A*. If *fact* = 'F' and *equed* is not 'N', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*.

The array *af* is an input argument if *fact* = 'F'. In this case it contains on entry the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization *A* = *P***L***U* as computed by p?getrf. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *A*.

The array *b* contains on entry the matrix *B* whose columns are the right-hand sides for the systems of equations.

work is a workspace array. The size of *work* is (*lwork*).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

iaf, *jaf*

(global) INTEGER. The row and column indices in the global matrix *AF* indicating the first row and the first column of the subarray *af(iaf:iaf+n-1, jaf:jaf+n-1)*, respectively.

descaf

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *AF*.

ib, *jb*

(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B(ib:ib+n-1, jb:jb+nrhs-1)*, respectively.

descb

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

ipiv

(local) INTEGER Array of size *LOCr(m_a)+mb_a*.

The array *ipiv* is an input argument if *fact = 'F'* .

On entry, it contains the pivot indices from the factorization *A = P*L*U* as computed by p?getrf; (local) row i of the matrix was interchanged with the (global) row *ipiv*(i).

This array must be aligned with *A(ia:ia+n-1, *)*.

equed

(global) CHARACTER*1. Must be 'N', 'R', 'C', or 'B'. *equed* is an input argument if *fact* = 'F' . It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N');

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by diag(*r*);

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by diag(*c*);

If *equed* = 'B', both row and column equilibration was done; *A* has been replaced by diag(*r*)**A**diag(*c*).

r, *c*

(local) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Arrays of size *LOCr*(m_a) and *LOCc*(n_a), respectively.

The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments. If *equed* = 'R' or 'B', *A* is multiplied on the left by diag(*r*); if *equed* = 'N' or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by diag(*c*); if *equed* = 'N' or 'R', *c* is not accessed.

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive. Array *r* is replicated in every process column, and is aligned with the distributed matrix *A*. Array *c* is replicated in every process row, and is aligned with the distributed matrix *A*.

ix, *jx*
(global) INTEGER. The row and column indices in the global matrix *X* indicating the first row and the first column of the submatrix *X*(*ix:ix+n-1*, *jx:jx+nrhs-1*), respectively.

descx
(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *X*.

lwork
(local or global) INTEGER. The size of the array *work* ; must be at least max(p?gecon(*lwork*), p?gerfs(*lwork*))+*LOCr*(n_a).

iwork
(local, psgesvx/pdgesvx only)INTEGER. Workspace array. The size of *iwork* is (*liwork*).

liwork
(local, psgesvx/pdgesvx only)INTEGER. The size of the array *iwork* , must be at least *LOCr*(n_a).

rwork
(local) REAL for pcgesvx

DOUBLE PRECISION for pzgesvx.

Workspace array, used in complex flavors only.

The size of *rwork* is (*lrwork*).

lrwork
(local or global, pcgesvx/pzgesvx only)INTEGER. The size of the array *rwork*;must be at least 2**LOCc*(n_a) .

Output Parameters

x
(local)

REAL for psgesvx

DOUBLE PRECISION for pdgesvx

COMPLEX for pcgesvx

DOUBLE COMPLEX for pzgesvx.

Pointer into the local memory to an array of local size *x*(lld_x,*LOCc*(*jx+nrhs-1*)).

If *info* = 0, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed*≠'N', and the solution to the *equilibrated* system is:

diag(*C*)-1**X*, if *trans* = 'N' and *equed* = 'C' or 'B'; and diag(*R*)-1**X*, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.

a
Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *equed*≠'N', *A* is scaled on exit as follows:

equed = 'R': *A* = diag(*R*)**A*

equed = 'C': *A* = *A**diag(*c*)

equed = 'B': *A* = diag(*R*)**A**diag(*c*)

af

If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization *A* = *P*L*U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b

Overwritten by diag(*R*)*B if *trans* = 'N' and *equed* = 'R' or 'B';

overwritten by diag(*c*)*B if *trans* = 'T' and *equed* = 'C' or 'B'; not changed if *equed* = 'N'.

r, *c*

These arrays are output arguments if *fact*≠'F'.

See the description of *r*, *c* in *Input Arguments* section.

rcond

(global)REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

ferr, *berr*

(local)REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays of size *LOCc*(n_b) each. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

Arrays *ferr* and *berr* are both replicated in every process row, and are aligned with the matrices *B* and *X*.

ipiv

If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A* = *P*L*U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed

If *fact*≠'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work(1)

If *info*=0, on exit *work*(1) returns the minimum value of *lwork* required for optimum performance.

iwork(1)

If *info*=0, on exit *iwork*(1) returns the minimum value of *liwork* required for optimum performance.

rwork(1)

If *info*=0, on exit *rwork*(1) returns the minimum value of *lrwork* required for optimum performance.

info

INTEGER. If *info*=0, the execution is successful.

info < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = −(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = −*i*. If *info* = *i*, and *i* ≤ *n*, then *U*(*i*,*i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed. If *info* = *i*, and *i* = *n* +1, then *U* is nonsingular, but *rcond* is less than

machine precision. The factorization has been completed, but the matrix is singular to working precision and the solution and error bounds have not been computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gbsv

Computes the solution to the system of linear equations with a general banded distributed matrix and multiple right-hand sides.

Syntax

```
call psgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
call pdgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
call pcgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
call pzgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
```

Include Files

Description

The p?gbsvroutine computes the solution to a real or complex system of linear equations

sub(*A*)**X* = sub(*B*),

where sub(*A*) = *A*(*1:n*, *ja:ja+n-1*) is an *n*-by-*n* real/complex general banded distributed matrix with *bwl* subdiagonals and *bwu* superdiagonals, and *X* and sub(*B*)= *B*(*ib:ib+n-1*, *1:rhs*) are *n*-by-*nrhs* distributed matrices.

The *LU* decomposition with partial pivoting and row interchanges is used to factor sub(*A*) as sub(*A*) = *P*L*U*Q*, where *P* and *Q* are permutation matrices, and *L* and *U* are banded lower and upper triangular matrices, respectively. The matrix *Q* represents reordering of columns for the sake of parallelism, while *P* represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

n — (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(*A*) (*n*≥ 0).

bwl — (global) INTEGER. The number of subdiagonals within the band of *A* (0≤ *bwl* ≤ *n-1*).

bwu — (global) INTEGER. The number of superdiagonals within the band of *A* (0≤ *bwu* ≤ *n-1*).

nrhs — (global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix sub(*B*) (*nrhs*≥ 0).

a, *b*

(local)

REAL for psgbsv

DOUBLE PRECISON for pdgbsv

COMPLEX for pcgbsv

DOUBLE COMPLEX for pzgbsv.

Pointers into the local memory to arrays of local size *a*(*lld_a*,*LOCc*(*ja*+*n*-1)) and *b*(*lld_b*,*LOCc*(*nrhs*)).

On entry, the array *a* contains the local pieces of the global array *A*.

On entry, the array *b* contains the right hand side distributed matrix sub(*B*).

ja

(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;

else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

ib

(global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;

else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

work

(local)

REAL for psgbsv

DOUBLE PRECISON for pdgbsv

COMPLEX for pcgbsv

DOUBLE COMPLEX for pzgbsv.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER. The size of the array *work*, must be at least

lwork ≥ (NB+*bwu*)*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2*bwu*) +

+ max(*nrhs* *(NB+2*bwl*+4*bwu*), 1).

Output Parameters

a

On exit, contains details of the factorization. Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

b	On exit, this array contains the local pieces of the solution distributed matrix *X*.
ipiv	(local) INTEGER array. The size of *ipiv* must be at least *desca*(NB). This array contains pivot indices for local factorizations. You should not alter the contents between factorization and solve.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*. *info*> 0: If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not nonsingular, and the factorization was not completed. If *info* = k > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dbsv

Solves a general band system of linear equations.

Syntax

```
call psdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pddbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pcdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pzdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Include Files

Description

The p?dbsvroutine solves the following system of linear equations:

A(1:*n*, *ja*:*ja*+*n*-1)* *X* = *B*(*ib*:*ib*+*n*-1, 1:*nrhs*),

where *A*(1:*n*, *ja*:*ja*+*n*-1) is an *n*-by-*n* real/complex banded diagonally dominant-like distributed matrix with bandwidth *bwl*, *bwu*.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into *LU*.

Input Parameters

n

(global) INTEGER. The order of the distributed submatrix *A*, (*n*≥ 0).

bwl

(global) INTEGER. Number of subdiagonals. 0 ≤ *bwl* ≤ *n*-1.

bwu

(global) INTEGER. Number of subdiagonals. 0 ≤ *bwu* ≤ *n*-1.

nrhs

(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix *B*, (*nrhs* ≥ 0).

a

(local). REAL for psdbsv

DOUBLE PRECISION for pddbsv

COMPLEX for pcdbsv

DOUBLE COMPLEX for pzdbsv.

Pointer into the local memory to an array with leading size *lld_a* ≥ (*bwl* +*bwu*+1) (stored in *desca*). On entry, this array contains the local pieces of the distributed matrix.

ja

(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen*.

If 1*d* type (*dtype_a*=501 or 502), *dlen* ≥ 7;

If 2*d* type (*dtype_a*=1), *dlen* ≥ 9.

The array descriptor for the distributed matrix *A*.

Contains information of mapping of *A* to memory.

b

(local)

REAL for psdbsv

DOUBLE PRECISON for pddbsv

COMPLEX for pcdbsv

DOUBLE COMPLEX for pzdbsv.

Pointer into the local memory to an array of local lead size *lld_b* ≥ *nb*. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib* +*n*-1, 1:*nrhs*).

ib

(global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

descb

(global and local) INTEGER array of size *dlen*.

If 1*d* type (*dtype_b* =502), *dlen* ≥ 7;

If 2*d* type (*dtype_b* =1), *dlen* ≥ 9.

The array descriptor for the distributed matrix *B*.

Contains information of mapping of *B* to memory.

work (local).

`REAL` for `psdbsv`

`DOUBLE PRECISON` for `pddbsv`

`COMPLEX` for `pcdbsv`

`DOUBLE COMPLEX` for `pzdbsv`.

Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) `INTEGER`. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*`(1)` and an error code is returned.

lwork ≥ *nb*(*bwl*+*bwu*)+6max(*bwl*,*bwu*)*max(*bwl*,*bwu*)
+max((max(*bwl*,*bwu*)*nrhs*), max(*bwl*,*bwu*)*max(*bwl*,*bwu*))

Output Parameters

a On exit, this array contains information containing details of the factorization.

Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

b On exit, this contains the local piece of the solutions distributed matrix *X*.

work On exit, *work*`(1)` contains the minimal *lwork*.

info (local) `INTEGER`. If *info*=0, the execution is successful.

< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

> 0: If *info* = *k* < NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dtsv

Solves a general tridiagonal system of linear equations.

Syntax

```
call psdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pddtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pcdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pzdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
```

Include Files

Description

The routine solves a system of linear equations

`A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),`

where `A(1:n, ja:ja+n-1)` is an *n*-by-*n* complex tridiagonal diagonally dominant-like distributed matrix.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into *L U*.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix *A*(*n*≥ 0).
nrhs	INTEGER. The number of right hand sides; the number of columns of the distributed matrix *B*(*nrhs*≥ 0).
dl	(local).REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*. Must be of size > *desca*(*nb_*).
d	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the main diagonal of the matrix.
du	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.
ja	(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array of size *dlen*.

If 1*d type* (*dtype_a*=501 or 502), *dlen* ≥ 7;

If 2*d type* (*dtype_a*=1), *dlen* ≥ 9.

The array descriptor for the distributed matrix *A*.

Contains information of mapping of *A* to memory.

b (local)

REAL for psdtsv

DOUBLE PRECISON for pddtsv

COMPLEX for pcdtsv

DOUBLE COMPLEX for pzdtsv.

Pointer into the local memory to an array of local lead size *lld_b* > *nb*. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib* +*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

descb (global and local) INTEGER array of size *dlen*.

If 1*d type* (*dtype_b* =502), *dlen* ≥ 7;

If 2*d type* (*dtype_b* =1), *dlen* ≥ 9.

The array descriptor for the distributed matrix *B*.

Contains information of mapping of *B* to memory.

work (local).

REAL for psdtsv

DOUBLE PRECISON for pddtsv

COMPLEX for pcdtsv

DOUBLE COMPLEX for pzdtsv. Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. *lwork* > (12*NPCOL+3**nb*)+max((10+2*min(100, *nrhs*))*NPCOL+4**nrhs*, 8*NPCOL)

Output Parameters

dl On exit, this array contains information containing the * factors of the matrix.

d On exit, this array contains information containing the * factors of the matrix. Must be of size > *desca*(*nb_*).

du On exit, this array contains information containing the * factors of the matrix. Must be of size > *desca*(*nb_*).

b	On exit, this contains the local piece of the solutions distributed matrix *X*.
work	On exit, *work*(1) contains the minimal *lwork*.
info	(local) INTEGER. If *info*=0, the execution is successful. < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. > 0: If *info* = *k*<NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed. If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?posv

Solves a symmetric positive definite system of linear equations.

Syntax

call psposv(*uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *info*)

call pdposv(*uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *info*)

call pcposv(*uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *info*)

call pzposv(*uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *info*)

Include Files

Description

The p?posvroutine computes the solution to a real/complex system of linear equations

sub(*A*)**X* = sub(*B*),

where sub(*A*) denotes *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1) and is an *n*-by-*n* symmetric/Hermitian distributed positive definite matrix and *X* and sub(*B*) denoting *B*(*ib*:*ib*+*n*-1,*jb*:*jb*+*nrhs*-1) are *n*-by-*nrhs* distributed matrices. The Cholesky decomposition is used to factor sub(*A*) as

sub(*A*) = U^T**U*, if *uplo* = 'U', or

sub(*A*) = *L**L^T, if *uplo* = 'L',

where *U* is an upper triangular matrix and *L* is a lower triangular matrix. The factored form of sub(*A*) is then used to solve the system of equations.

Input Parameters

uplo	(global) CHARACTER. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of sub(*A*) is stored.
n	(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥ 0).

nrhs INTEGER. The number of right-hand sides; the number of columns of the distributed matrix sub(*B*) (*nrhs*≥ 0).

a (local)

REAL for psposv

DOUBLE PRECISION for pdposv

COMPLEX for pcposv

COMPLEX*16 for pzposv.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*) to be factored.

If *uplo* = '*U*', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If *uplo* = '*L*', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

b (local)

REAL for psposv

DOUBLE PRECISON for pdposv

COMPLEX for pcposv

COMPLEX*16 for pzposv.

Pointer into the local memory to an array of size (*lld_b*, *LOCc*(*jb* +*nrhs*-1)). On entry, the local pieces of the right hand sides distributed matrix sub(*B*).

ib, *jb* (global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Output Parameters

a On exit, if *info* = 0, this array contains the local pieces of the factor *U* or *L* from the Cholesky factorization sub(*A*) = U^H*U, or *L*L^H.

b On exit, if *info* = 0, sub(*B*) is overwritten by the solution distributed matrix *X*.

info (global) INTEGER.

If *info* =0, the execution is successful.

If *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* > 0: If *info* = *k*, the leading minor of order *k*, A(*ia*:*ia*+*k*-1, *ja*:*ja*+*k*-1) is not positive definite, and the factorization could not be completed, and the solution has not been computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?posvx

Solves a symmetric or Hermitian positive definite system of linear equations.

Syntax

call psposvx(*fact*, *uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *equed*, *sr*, *sc*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pdposvx(*fact*, *uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *equed*, *sr*, *sc*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pcposvx(*fact*, *uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *equed*, *sr*, *sc*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pzposvx(*fact*, *uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *equed*, *sr*, *sc*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

Include Files

Description

The p?posvxroutine uses the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ to compute the solution to a real or complex system of linear equations

A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1)**X* = *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*nrhs*-1),

where A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) is a *n*-by-*n* matrix and *X* and *B*(*ib*:*ib*+*n*-1,*jb*:*jb*+*nrhs*-1) are *n*-by-*nrhs* matrices.

Error bounds on the solution and a condition estimate are also provided.

In the following comments *y* denotes *Y*(*iy*:*iy*+*m*-1, *jy*:*jy*+*k*-1), an *m*-by-*k* matrix where *y* can be *a*, *af*, *b* and *x*.

The routine p?posvx performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

   diag(*sr*)**A**diag(*sc*)*inv(diag(*sc*))**X* = diag(*sr*)**B*

   Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by diag(*sr*)**A**diag(*sc*) and *B* by diag(*sr*)**B* .
2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

   $A = U^T*U$, if *uplo* = 'U', or

   $A = L*L^T$, if *uplo* = 'L',

   where *U* is an upper triangular matrix and *L* is a lower triangular matrix.

3. The factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, steps 4-6 are skipped
4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by diag(*sr*) so that it solves the original system before equilibration.

Input Parameters

fact

(global) CHARACTER. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F': on entry, *af* contains the factored form of *A*. If *equed* = 'Y', the matrix *A* has been equilibrated with scaling factors given by *s*. *a* and *af* will not be modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.

If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

uplo

(global) CHARACTER. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored.

n

(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥ 0).

nrhs

(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrices *B* and *X*. (*nrhs*≥ 0).

a

(local)

REAL for psposvx

DOUBLE PRECISION for pdposvx

COMPLEX for pcposvx

DOUBLE COMPLEX for pzposvx.

Pointer into the local memory to an array of local size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the symmetric/Hermitian matrix *A*, except if *fact* = 'F' and *equed* = 'Y', then *A* must contain the equilibrated matrix diag(*sr*)*A*diag(*sc*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced. *A* is not modified if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N' on exit.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

af (local)

REAL for psposvx

DOUBLE PRECISION for pdposvx

COMPLEX for pcposvx

DOUBLE COMPLEX for pzposvx.

Pointer into the local memory to an array of local size (*lld_af*, *LOCc*(*ja*+*n*-1)).

If *fact* = 'F', then *af* is an input argument and on entry contains the triangular factor *U* or *L* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$, in the same storage format as *A*. If *equed* ≠ 'N', then *af* is the factored form of the equilibrated matrix diag(*sr*)*A*diag(*sc*).

iaf, *jaf* (global) INTEGER. The row and column indices in the global matrix *AF* indicating the first row and the first column of the submatrix *AF*, respectively.

descaf (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *AF*.

equed (global) CHARACTER. Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N');

If *equed* = 'Y', equilibration was done and *A* has been replaced by diag(*sr*)*A*diag(*sc*).

sr (local)

REAL for psposvx

DOUBLE PRECISION for pdposvx

COMPLEX for pcposvx

DOUBLE COMPLEX for pzposvx.

Array of size *lld_a*.

The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *equed* = 'N', *s* is not accessed.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

b (local)

REAL for psposvx

DOUBLE PRECISION for pdposvx

COMPLEX for pcposvx

DOUBLE COMPLEX for pzposvx.

Pointer into the local memory to an array of local size (*lld_b*, *LOCc*(*jb*+*nrhs*-1)). On entry, the *n*-by-*nrhs* right-hand side matrix *B*.

ib, *jb*

(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

descb

(global and local) INTEGER. Array of size *dlen_*. The array descriptor for the distributed matrix *B*.

x

(local)

REAL for psposvx

DOUBLE PRECISION for pdposvx

COMPLEX for pcposvx

DOUBLE COMPLEX for pzposvx.

Pointer into the local memory to an array of local size (*lld_x*, *LOCc*(*jx*+*nrhs*-1)).

ix, *jx*

(global) INTEGER. The row and column indices in the global matrix *X* indicating the first row and the first column of the submatrix *X*, respectively.

descx

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *X*.

work

(local)

REAL for psposvx

DOUBLE PRECISION for pdposvx

COMPLEX for pcposvx

DOUBLE COMPLEX for pzposvx.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*. *lwork* is local input and must be at least *lwork* = max(p?pocon(*lwork*), p?porfs(*lwork*)) + *LOCr*(*n_a*).

lwork = 3**desca*(*lld_*).

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

iwork

(local) INTEGER. Workspace array of size *liwork*.

liwork

(local or global)

INTEGER. The size of the array *iwork*. *liwork* is local input and must be at least *liwork* = *desca*(*lld_*)*liwork* = *LOCr*(*n_a*).

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, if *fact* = 'E' and *equed* = 'Y', *a* is overwritten by diag(*sr*)*a*diag(*sc*).

af

If *fact* = 'N', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$ of the original matrix *A*.

If *fact* = 'E', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$ of the equilibrated matrix *A* (see the description of *A* for the form of the equilibrated matrix).

equed

If *fact*≠'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

sr

This array is an output argument if *fact*≠'F'.

See the description of *sr* in *Input Arguments* section.

sc

This array is an output argument if *fact*≠'F'.

See the description of *sc* in *Input Arguments* section.

b

On exit, if *equed* = 'N', *b* is not modified; if *trans* = 'N' and *equed* = 'R' or 'B', *b* is overwritten by diag(*r*)*b; if *trans* = 'T' or 'C' and *equed* = 'C' or 'B', *b* is overwritten by diag(*c*)*b.

x

(local)

REAL for psposvx

DOUBLE PRECISION for pdposvx

COMPLEX for pcposvx

DOUBLE COMPLEX for pzposvx.

If *info* = 0 the *n*-by-*nrhs* solution matrix *X* to the original system of equations.

Note that *A* and *B* are modified on exit if *equed*≠'N', and the solution to the equilibrated system is

inv(diag(*sc*))*X if *trans* = 'N' and *equed* = 'C' or 'B', or

inv(diag(*sr*))*X if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.

rcond

(global)

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond*=0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays of size at least max(*LOC*, *n_b*). The estimated forward error bounds for each solution vector *X*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution, *ferr*(*j*) bounds the magnitude of the largest entry in (*X*(*j*) – *xtrue*) divided by the magnitude of the largest entry in *X*(*j*). The quality of the error bound depends on the quality of the estimate of norm(inv(*A*)) computed in the code; if the estimate of norm(inv(*A*)) is accurate, the error bound is guaranteed.

berr (local)

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays of size at least max(*LOC*, *n_b*). The componentwise relative backward error of each solution vector *X*(*j*) (the smallest relative change in any entry of *A* or *B* that makes *X*(*j*) an exact solution).

work(1) (local) On exit, *work*(1) returns the minimal and optimal *liwork*.

info (global) INTEGER.

If *info*=0, the execution is successful.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value

> 0: if *info* = *i*, and *i* is ≤ *n*: if *info* = *i*, the leading minor of order *i* of *a* is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed.

= *n*+1: *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision, and the solution and error bounds have not been computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?pbsv

Solves a symmetric/Hermitian positive definite banded system of linear equations.

Syntax

```
call pspbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pdpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pcpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pzpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Include Files

Description

The p?pbsvroutine solves a system of linear equations

A(1:*n*, *ja*:*ja*+*n*-1)***X* = *B*(*ib*:*ib*+*n*-1, 1:*nrhs*),

where A(1:n, ja:ja+n-1) is an *n*-by-*n* real/complex banded symmetric positive definite distributed matrix with bandwidth *bw*.

Cholesky factorization is used to factor a reordering of the matrix into *L***L'*.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

uplo

(global) CHARACTER. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular of *A* is stored.

If *uplo* = 'U', the upper triangular *A* is stored

If *uplo* = 'L', the lower triangular of *A* is stored.

n

(global) INTEGER. The order of the distributed matrix *A* (*n*≥ 0).

bw

(global) INTEGER. The number of subdiagonals in *L* or *U*. 0 ≤ *bw* ≤ *n*-1.

nrhs

(global) INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs*≥ 0).

a

(local). REAL for pspbsv

DOUBLE PRECISON for pdpbsv

COMPLEX for pcpbsv

DOUBLE COMPLEX for pzpbsv.

Pointer into the local memory to an array with leading size *lld_a* ≥ (*bw* +1) (stored in *desca*). On entry, this array contains the local pieces of the distributed matrix sub(*A*) to be factored.

ja

(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

b

(local)

REAL for pspbsv

DOUBLE PRECISON for pdpbsv

COMPLEX for pcpbsv

DOUBLE COMPLEX for pzpbsv.

Pointer into the local memory to an array of local lead size *lld_b* ≥ *nb*. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib* +*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

descb (global and local) INTEGER array of size *dlen*.

If 1D type (*dtype_b* =502), *dlen* ≥ 7;

If 2D type (*dtype_b* =1), *dlen* ≥ 9.

The array descriptor for the distributed matrix *B*.

Contains information of mapping of *B* to memory.

work (local).

REAL for pspbsv

DOUBLE PRECISON for pdpbsv

COMPLEX for pcpbsv

DOUBLE COMPLEX for pzpbsv.

Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. *lwork* ≥ (*nb*+2*bw*)*bw* +max((*bw*nrhs*), *bw*bw*)

Output Parameters

a On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

b On exit, contains the local piece of the solutions distributed matrix *X*.

work On exit, *work*(1) contains the minimal *lwork*.

info (global) INTEGER. If *info*=0, the execution is successful.

< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i*100+j)*, if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

> 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ptsv

Syntax

Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations.

```
call psptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pdptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pcptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pzptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
```

Include Files

Description

The p?ptsvroutine solves a system of linear equations

A(1:n, ja:ja+n-1)*X = B(ib:ib+n-1, 1:nrhs),

where A(1:n, ja:ja+n-1) is an *n*-by-*n* real tridiagonal symmetric positive definite distributed matrix.

Cholesky factorization is used to factor a reordering of the matrix into *L***L*'.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

n	(global) INTEGER. The order of matrix *A* (*n*≥ 0).
nrhs	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix *B* (*nrhs*≥ 0).
d	(local) REAL for psptsv DOUBLE PRECISON for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Pointer to local part of global vector storing the main diagonal of the matrix.
e	(local) REAL for psptsv DOUBLE PRECISON for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, *du*(n) is not referenced, and *du* must be aligned with *d*.
ja	(global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array of size *dlen*.

If 1*d* type (*dtype_a*=501 or 502), *dlen* ≥ 7;

If 2*d* type (*dtype_a*=1), *dlen* ≥ 9.

The array descriptor for the distributed matrix *A*.

Contains information of mapping of *A* to memory.

b (local)

REAL for psptsv

DOUBLE PRECISON for pdptsv

COMPLEX for pcptsv

DOUBLE COMPLEX for pzptsv.

Pointer into the local memory to an array of local lead size *lld_b* ≥ *nb*.

On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

descb (global and local) INTEGER array of size *dlen*.

If 1*d* type (*dtype_b* = 502), *dlen* ≥ 7;

If 2*d* type (*dtype_b* = 1), *dlen* ≥ 9.

The array descriptor for the distributed matrix *B*.

Contains information of mapping of *B* to memory.

work (local).

REAL for psptsv

DOUBLE PRECISON for pdptsv

COMPLEX for pcptsv

DOUBLE COMPLEX for pzptsv.

Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. *lwork* > (12*NPCOL+3**nb*)+max((10+2*min(100, *nrhs*))*NPCOL+4**nrhs*, 8*NPCOL).

Output Parameters

d On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to *desca*(*nb_*).

e On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to *desca*(*nb_*).

b On exit, this contains the local piece of the solutions distributed matrix *X*.

work	On exit, *work*(1) contains the minimal *lwork*.
info	(local) INTEGER. If *info*=0, the execution is successful. < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. > 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed. If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gels

Solves overdetermined or underdetermined linear systems involving a matrix of full rank.

Syntax

```
call psgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
call pdgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
call pcgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
call pzgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
```

Include Files

Description

The p?gels routine solves overdetermined or underdetermined real/ complex linear systems involving an *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), or its transpose/ conjugate-transpose, using a *QTQ* or *LQ* factorization of sub(*A*). It is assumed that sub(*A*) has full rank.

The following options are provided:

1. If *trans* = 'N' and *m*≥*n*: find the least squares solution of an overdetermined system, that is, solve the least squares problem

   minimize ||sub(*B*) - sub(*A*)**X*||
2. If *trans* = 'N' and *m* < *n*: find the minimum norm solution of an underdetermined system sub(*A*)**X* = sub(*B*).
3. If *trans* = 'T' and *m*≥*n*: find the minimum norm solution of an undetermined system sub(*A*)^T**X* = sub(*B*).
4. If *trans* = 'T' and *m* < *n*: find the least squares solution of an overdetermined system, that is, solve the least squares problem

   minimize ||sub(*B*) - sub(*A*)^T**X*||,

   where sub(*B*) denotes *B*(*ib*:*ib*+*m*-1, *jb*:*jb*+*nrhs*-1) when *trans* = 'N' and *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*nrhs*-1) otherwise. Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; when *trans* = 'N', the solution vectors are stored as the columns of the *n*-by-*nrhs* right hand side matrix sub(*B*) and the *m*-by-*nrhs* right hand side matrix sub(*B*) otherwise.

Input Parameters

trans

(global) CHARACTER. Must be 'N', or 'T'.

If *trans* = 'N', the linear system involves matrix sub(*A*);

If *trans* = 'T', the linear system involves the transposed matrix A^T (for real flavors only).

m

(global) INTEGER. The number of rows in the distributed matrix sub (*A*) (*m*≥ 0).

n

(global) INTEGER. The number of columns in the distributed matrix sub (*A*) (*n*≥ 0).

nrhs

(global) INTEGER. The number of right-hand sides; the number of columns in the distributed submatrices sub(*B*) and *X*. (*nrhs*≥ 0).

a

(local)

REAL for psgels

DOUBLE PRECISION for pdgels

COMPLEX for pcgels

DOUBLE COMPLEX for pzgels.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, contains the *m*-by-*n* matrix *A*.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

b

(local)

REAL for psgels

DOUBLE PRECISION for pdgels

COMPLEX for pcgels

DOUBLE COMPLEX for pzgels.

Pointer into the local memory to an array of local size (*lld_b*, *LOCc*(*jb* +*nrhs*-1)). On entry, this array contains the local pieces of the distributed matrix *B* of right-hand side vectors, stored columnwise; sub(*B*) is *m*-by-*nrhs* if *trans*='N', and *n*-by-*nrhs* otherwise.

ib, *jb*

(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

descb

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

work

(local)

REAL for psgels

DOUBLE PRECISION for pdgels

COMPLEX for pcgels

DOUBLE COMPLEX for pzgels.

Workspace array with size *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*|*lwork* is local input and must be at least $lwork \geq ltau + \max(lwf, lws)$, where if $m > n$, then

$ltau = \text{numroc}(ja+\min(m,n)-1, nb\_a, \text{MYCOL}, csrc\_a, \text{NPCOL})$,

$lwf = nb\_a*(mpa0 + nqa0 + nb\_a)$

$lws = \max((nb\_a*(nb\_a-1))/2, (nrhsqb0 + mpb0)*nb\_a) + nb\_a*nb\_a$

else

$ltau = \text{numroc}(ia+\min(m,n)-1, mb\_a, \text{MYROW}, rsrc\_a, \text{NPROW})$,

$lwf = mb\_a * (mpa0 + nqa0 + mb\_a)$

$lws = \max((mb\_a*(mb\_a-1))/2, (npb0 + \max(nqa0 + \text{numroc}(\text{numroc}(n+iroffb, mb\_a, 0, 0, \text{NPROW}), mb\_a, 0, 0, lcmp), nrhsqb0))*mb\_a) + mb\_a*mb\_a$

end if,

where $lcmp = lcm/\text{NPROW}$ with $lcm = \text{ilcm}(\text{NPROW}, \text{NPCOL})$,

$iroffa = \text{mod}(ia-1, mb\_a)$,

$icoffa = \text{mod}(ja-1, nb\_a)$,

$iarow = \text{indxg2p}(ia, mb\_a, \text{MYROW}, rsrc\_a, \text{NPROW})$,

$iacol = \text{indxg2p}(ja, nb\_a, \text{MYROW}, rsrc\_a, \text{NPROW})$

$mpa0 = \text{numroc}(m+iroffa, mb\_a, \text{MYROW}, iarow, \text{NPROW})$,

$nqa0 = \text{numroc}(n+icoffa, nb\_a, \text{MYCOL}, iacol, \text{NPCOL})$,

$iroffb = \text{mod}(ib-1, mb\_b)$,

$icoffb = \text{mod}(jb-1, nb\_b)$,

$ibrow = \text{indxg2p}(ib, mb\_b, \text{MYROW}, rsrc\_b, \text{NPROW})$,

$ibcol = \text{indxg2p}(jb, nb\_b, \text{MYCOL}, csrc\_b, \text{NPCOL})$,

$mpb0 = \text{numroc}(m+iroffb, mb\_b, \text{MYROW}, icrow, \text{NPROW})$,

$nqb0 = \text{numroc}(n+icoffb, nb\_b, \text{MYCOL}, ibcol, \text{NPCOL})$,

NOTE
$\text{mod}(x,y)$ is the integer remainder of x/y.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW, and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, If $m \geq n$, sub(*A*) is overwritten by the details of its *QR* factorization as returned by p?geqrf; if $m < n$, sub(*A*) is overwritten by details of its *LQ* factorization as returned by p?gelqf.

b

On exit, sub(*B*) is overwritten by the solution vectors, stored columnwise: if *trans* = 'N' and $m \geq n$, rows 1 to *n* of sub(*B*) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $n+1$ to *m* in that column;

If *trans* = 'N' and $m < n$, rows 1 to *n* of sub(*B*) contain the minimum norm solution vectors;

If *trans* = 'T' and $m \geq n$, rows 1 to *m* of sub(*B*) contain the minimum norm solution vectors; if *trans* = 'T' and $m < n$, rows 1 to *m* of sub(*B*) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $m+1$ to *n* in that column.

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?syev

Computes selected eigenvalues and eigenvectors of a symmetric matrix.

Syntax

```
call pssyev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, info)
call pdsyev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, info)
```

Include Files

Description

The p?syevroutine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* by calling the recommended sequence of ScaLAPACK routines.

In its present form, the routine assumes a homogeneous system and makes no checks for consistency of the eigenvalues or eigenvectors across the different processes. Because of this, it is possible that a heterogeneous system may return incorrect results without any error messages.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz — (global) CHARACTER. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors:

If *jobz* ='N', then only eigenvalues are computed.

If *jobz* ='V', then eigenvalues and eigenvectors are computed.

uplo — (global) CHARACTER. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n — (global) INTEGER. The number of rows and columns of the matrix *A* (*n*≥ 0).

a — (local)

REAL for pssyev.

DOUBLE PRECISION for pdsyev.

Block cyclic array of global size (*n*, *n*) and local size (*lld_a*, *LOCc*(*ja* +*n*-1)). On entry, the symmetric matrix *A*.

If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the symmetric matrix.

If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the symmetric matrix.

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

iz, *jz* — (global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*.

work — (local)

REAL for pssyev.

DOUBLE PRECISION for pdsyev.

Array of size *lwork*.

lwork — (local) INTEGER. See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then *lwork* ≥ 5*n* + *sizesytrd* + 1,

where *sizesytrd*is the workspace for p?sytrd and is max(NB*(*np* +1), 3*NB).

If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is:

qrmem = 2*n-2

lwmin = 5*n + n*ldc + max(*sizemqrleft*, *qrmem*) + 1

Variable definitions:

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);

nn = max(*n*, *nb*, 2);

desca(*rsrc_*) = *desca*(*rsrc_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0

np = numroc(*nn*, *nb*, 0, 0, NPROW)

nq = numroc(max(*n*, *nb*, 2), *nb*, 0, 0, NPCOL)

nrc = numroc(*n*, *nb*, *myprowc*, 0, NPROCS)

ldc = max(1, *nrc*)

sizemqrleft is the workspace for p?ormtr when its *side* argument is 'L'.

myprowc is defined when a new context is created as follows:

call blacs_get(*desca*(*ctxt_*), 0, *contextc*)

call blacs_gridinit(*contextc*, 'R', NPROCS, 1)

call blacs_gridinfo(*contextc*, *nprowc*, *npcolc*, *myprowc*, *mypcolc*)

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, the lower triangle (if *uplo*='L') or the upper triangle (if *uplo*='U') of *A*, including the diagonal, is destroyed.

w

(global). REAL for pssyev

DOUBLE PRECISION for pdsyev

Array of size *n*.

On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z

(local). REAL for pssyev

DOUBLE PRECISION for pdsyev

Array, global size (*n*, *n*), local size (*lld_z*, *LOCc*(*jz*+*n*-1)). If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues.

If *jobz* = 'N', then *z* is not referenced.

work(1)

On output, *work*(1) returns the workspace needed to guarantee completion. If the input parameters are incorrect, *work*(1) may also be incorrect.

If *jobz* = 'N'*work*(1) = minimal (optimal) amount of workspace

If *jobz* = 'V'*work*(1) = minimal workspace required to generate all the eigenvectors.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* > 0:

If *info*= 1 through *n*, the *i*-th eigenvalue did not converge in ?steqr2 after a total of 30*n* iterations.

If *info*= *n*+1, then p?syev has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?syev cannot be guaranteed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?syevd

Computes all eigenvalues and eigenvectors of a real symmetric matrix by using a divide and conquer algorithm.

Syntax

call pssyevd(*jobz*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *w*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pdsyevd(*jobz*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *w*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

Include Files

Description

The p?syevd routine computes all eigenvalues and eigenvectors of a real symmetric matrix *A* by using a divide and conquer algorithm.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz

(global) CHARACTER*1. Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* (*n*≥ 0).

a (local).

REAL for pssyevd

DOUBLE PRECISION for pdsyevd.

Block cyclic array of global size (*n*, *n*) and local size (*lld_a*, *LOCc*(*ja* +*n*-1)). On entry, the symmetric matrix *A*.

If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the symmetric matrix.

If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the symmetric matrix.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*. If *desca*(*ctxt_*) is incorrect, p?syevd cannot guarantee correct error reporting.

iz, *jz* (global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local).

REAL for pssyevd

DOUBLE PRECISION for pdsyevd.

Array of size *lwork*.

lwork (local) INTEGER. The size of the array *work*.

If eigenvalues are requested:

lwork≥ max(1+6**n* + 2**np***nq*, *trilwmin*) + 2**n*

with *trilwmin* = 3**n* + max(*nb***(*np* + 1), 3**nb*)

np = numroc(*n*, *nb*, *myrow*, *iarow*, NPROW)

nq = numroc(*n*, *nb*, *mycol*, *iacol*, NPCOL)

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array of size *liwork*.

liwork (local) INTEGER , size of *iwork*.

liwork = 7*n + 8*npcol + 2.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L'), or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w (global).

REAL for pssyevd

DOUBLE PRECISION for pdsyevd.

Array of size *n*. If *info* = 0, *w* contains the eigenvalues in the ascending order.

z (local).

REAL for pssyevd

DOUBLE PRECISION for pdsyevd.

Array, global size (*n*, *n*), local size (*lld_z*, *LOCc*(*jz*+*n*-1)).

The *z* parameter contains the orthonormal eigenvectors of the matrix *A*.

work(1) On exit, returns adequate workspace to allow optimal performance.

iwork(1) (local).

On exit, if *liwork* > 0, *iwork*(1) returns the optimal *liwork*.

info (global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*). If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info*> 0:

The algorithm failed to compute the *info*/(*n*+1)-th eigenvalue while working on the submatrix lying in global rows and columns mod(*info*,*n* +1).

NOTE

mod(*x*,*y*) is the integer remainder of *x*/*y*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using Relatively Robust Representation.

Syntax

```
call pssyevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z, iz, jz, descz, work, lwork, iwork, liwork, info )
```

```
call pdsyevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z, iz, jz, descz, work, lwork, iwork, liwork, info )
```

Include Files

Description

`p?syevr` computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* distributed in 2D blockcyclic format by calling the recommended sequence of ScaLAPACK routines.

First, the matrix *A* is reduced to real symmetric tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in `w`. The eigenvector matrix `z` is stored in 2D block-cyclic format distributed over all processors.

Note that subsets of eigenvalues/vectors can be selected by specifying a range of values or a range of indices for the desired eigenvalues.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

Input Parameters

jobz (global) CHARACTER*1

Specifies whether or not to compute the eigenvectors:

= 'N': Compute eigenvalues only.

= 'V': Compute eigenvalues and eigenvectors.

range (global) CHARACTER*1

= 'A': all eigenvalues will be found.

= 'V': all eigenvalues in the interval [`vl`,`vu`] will be found.

= 'I': the `il`-th through `iu`-th eigenvalues will be found.

uplo (global) CHARACTER*1

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n (global) INTEGER

The number of rows and columns of the matrix *a*. *n*≥ 0

a REAL for pssyevr

DOUBLE PRECISION for pdsyevr

Block cyclic array of global size (*n*, *n*), local size (*lld_a*, LOC_c(*ja*+*n*-1)).

This array contains the local pieces of the symmetric distributed matrix *A*. If *uplo* = 'U', only the upper triangular part of *a* is used to define the elements of the symmetric matrix. If *uplo* = 'L', only the lower triangular part of *a* is used to define the elements of the symmetric matrix.

On exit, the lower triangle (if *uplo*='L') or the upper triangle (if *uplo*='U') of *a*, including the diagonal, is destroyed.

ia (global) INTEGER

Global row index in the global matrix *A* that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

ja (global) INTEGER

Global column index in the global matrix *A* that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

desca (global and local) INTEGER array of size *dlen_*=9.

The array descriptor for the distributed matrix *a*.

vl REAL for pssyevr

DOUBLE PRECISION for pdsyevr

(global)

If *range*='V', the lower bound of the interval to be searched for eigenvalues. Not referenced if *range* = 'A' or 'I'.

vu REAL for pssyevr

DOUBLE PRECISION for pdsyevr

(global)

If *range*='V', the upper bound of the interval to be searched for eigenvalues. Not referenced if *range* = 'A' or 'I'.

il (global) INTEGER

If *range*='I', the index (from smallest to largest) of the smallest eigenvalue to be returned. *il*≥ 1.

Not referenced if *range* = 'A'.

iu (global) INTEGER

If *range*='I', the index (from smallest to largest) of the largest eigenvalue to be returned. min(*il*,*n*) ≤*iu*≤*n*.

Not referenced if *range* = 'A'.

iz

(global) INTEGER

Global row index in the global matrix *Z* that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

jz

(global) INTEGER

Global column index in the global matrix *Z* that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

descz

INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *z*.

The context *descz*(*ctxt_*) must equal *desca*(*ctxt_*). Also note the array alignment requirements specified below.

work

REAL for pssyevr

DOUBLE PRECISION for pdsyevr

(local workspace) array of size *lwork*

lwork

(local) INTEGER

Size of *work*, must be at least 3.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N') then

lwork≥ 2 + 5*n + max(12 * *nn*, *neig* * (*np0* + 1))

If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required is:

lwork≥ 2 + 5*n + max(18**nn*, *np0* * *mq0* + 2 * *neig* * *neig*) + (2 + iceil(*neig*, *nprow***npcol*))**nn*

Variable definitions:

neig = number of eigenvectors requested

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*)

nn = max(*n*, *neig*, 2)

desca(*rsrc_*) = *desca*(*csrc_nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0

np0 = numroc(*nn*, *neig*, 0, 0, *nprow*)

mq0 = numroc(max(*neig*, *neig*, 2), *neig*, 0, 0, *npcol*)

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*), and *nprow* and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

liwork (local) INTEGER

size of *iwork*

Let *nnp* = max(*n*, *nprow***npcol* + 1, 4). Then:

liwork≥ 12**nnp* + 2**n* when the eigenvectors are desired

liwork≥ 10**nnp* + 2**n* when only the eigenvalues have to be computed

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

OUTPUT Parameters

m (global) INTEGER

Total number of eigenvalues found. 0 ≤*m*≤*n*.

nz (global) INTEGER

Total number of eigenvectors computed. 0 ≤*nz*≤*m*.

The number of columns of *z* that are filled.

If *jobz*≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m*

w REAL for pssyevr

DOUBLE PRECISION for pdsyevr

(global) array of size *n*

Upon successful exit, the first *m* entries contain the selected eigenvalues in ascending order.

z REAL for pssyevr

DOUBLE PRECISION for pdsyevr

Block-cyclic array, global size(*n*, *n*), local size (*lld_z*, $LOC_c(jz+n-1)$) .

On exit, contains local pieces of distributed matrix *Z*.

work On return, *work*(1) contains the optimal amount of workspace required for efficient execution. If *jobz*='N' *work*(1) = optimal amount of workspace required to compute the eigenvalues. If *jobz*='V' *work*(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors.

iwork (local workspace) INTEGER array

On return, *iwork*(1) contains the amount of integer workspace required.

info (global) INTEGER

= 0: successful exit

< 0: If the *i*-th argument is an array and the *j*th-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The distributed submatrices *a*(*ia*:*, *ja*:*) and *z*(*iz*:*iz*+*m*-1,*jz*:*jz*+*n*-1) must satisfy the following alignment properties:

1. Identical (quadratic) dimension: *desca*(*m_*) = *descz*(*m_*) = *desca*(*n_*) = *descz*(*n_*)
2. Quadratic conformal blocking: *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*), *desca*(*rsrc_*) = *descz*(*rsrc_*)
3. mod(*ia*-1, *mb_a*) = mod(*iz*-1, *mb_z*) = 0

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

```
call pssyevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info)
```

```
call pdsyevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info)
```

Include Files

Description

The p?syevxroutine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) CHARACTER*1. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors:

If *jobz* ='N', then only eigenvalues are computed.

If *jobz* ='V', then eigenvalues and eigenvectors are computed.

range (global) CHARACTER*1. Must be 'A', 'V', or 'I'.

If *range* = 'A', all eigenvalues will be found.

If *range* = 'V', all eigenvalues in the half-open interval [*vl*, *vu*] will be found.

If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* (*n* ≥ 0).

a (local). REAL for pssyevx

DOUBLE PRECISION for pdsyevx.

Block cyclic array of global size (*n*, *n*) and local size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the symmetric matrix *A*.

If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the symmetric matrix.

If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the symmetric matrix.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

vl, *vu* (global)

REAL for pssyevx

DOUBLE PRECISION for pdsyevx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; *vl* ≤ *vu*. Not referenced if *range* = 'A' or 'I'.

il, *iu* (global) INTEGER.

If *range* ='I', the indices of the smallest and largest eigenvalues to be returned.

Constraints: *il* ≥ 1

min(*il*,*n*) ≤ *iu* ≤ *n*

Not referenced if *range* = 'A' or 'V'.

abstol (global).REAL for pssyevx

DOUBLE PRECISION for pdsyevx.

If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*, *b*] of width less than or equal to

abstol + *eps* * max(|*a*|,|*b*|),

where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S') not zero. If this routine returns with (mod(*info*,2) ≠ 0) or (mod(*info*/8,2) ≠ 0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

orfac

(global).REAL for pssyevx

DOUBLE PRECISION for pdsyevx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol=orfac*norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, *jz*

(global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work

(local)

REAL for pssyevx.

DOUBLE PRECISION for pdsyevx.

Array of size *lwork*.

lwork

(local) INTEGER. The size of the array *work*.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then *lwork* ≥ 5*n* + max(5*nn*, NB*(*np*0 + 1)).

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

lwork ≥ 5*n* + max(5*nn*, *np*0*mq*0 + 2*NB*NB) + iceil(*neig*, NPROW*NPCOL)**nn*

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to *lwork*:

(*clustersize*-1)**n*,

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

{*w*(*k*),..., *w*(*k*+*clustersize*-1)|*w*(*j*+1) ≤ *w*(*j*)) + *orfac**2*norm(*A*)},

where

neig = number of eigenvectors requested

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);

nn = max(*n*, *nb*, 2);

desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0;

*np*0 = numroc(*nn*, *nb*, 0, 0, NPROW);

*mq*0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL)

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

If *lwork* is too small to guarantee orthogonality, p?syevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, p?sygvx computes the eigenvalues and as many eigenvectors as possible.

Relationship between workspace, orthogonality & performance:

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

lwork ≥ max(*lwork*, 5**n* + *nsytrd_lwopt*),

where *lwork*, as defined previously, depends upon the number of eigenvectors requested, and

nsytrd_lwopt = *n* + 2*(*anb*+1)*(4**nps*+2) + (*nps* + 3)**nps;*

anb = pjlaenv(*desca*(*ctxt_*), 3, 'p?syttrd', 'L', 0, 0, 0, 0);

sqnpc = int(*sqrt*(*dble*(NPROW * NPCOL)));

nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*);

numroc is a ScaLAPACK tool functions;

pjlaenv is a ScaLAPACK environmental inquiry function

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

For large n, no extra workspace is needed, however the biggest boost in performance comes for small n, so it is wise to provide the extra workspace (typically less than a megabyte per process).

If *clustersize* > *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, *clustersize* = *n*-1) p?stein will perform no better than ?stein on single processor.

For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize*>*n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, size of *iwork*. *liwork* ≥ 6**nnp*

Where: *nnp* = max(*n*, NPROW*NPCOL + 1, 4)

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U')of *A*, including the diagonal, is overwritten.

m (global) INTEGER. The total number of eigenvalues found; 0 ≤ *m* ≤ *n*.

nz (global) INTEGER. Total number of eigenvectors computed. 0 ≤ *nz* ≤ *m*.

The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p?syevx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*≤*descz*(*n*_)) and sufficient workspace to compute them. (See *lwork*). p?syevx is always able to detect insufficient space without computation unless *range* = 'V'.

w (global).REAL for pssyevx

DOUBLE PRECISION for pdsyevx.

Array of size *n*. The first *m* elements contain the selected eigenvalues in ascending order.

z (local).REAL for pssyevx

DOUBLE PRECISION for pdsyevx.

Array, global size (*n*, *n*), local size `(lld_z, LOCc(jz+n-1))`.

If `jobz = 'V'`, then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If `jobz = 'N'`, then *z* is not referenced.

work(1)

On exit, returns workspace adequate workspace to allow optimal performance.

iwork(1)

On return, *iwork*(1) contains the amount of integer workspace required.

ifail

(global)INTEGER.

Array of size *n*.

If `jobz = 'V'`, then on normal exit, the first *m* elements of *ifail* are zero. If `(mod(info,2) ≠ 0)` on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.

If `jobz = 'N'`, then *ifail* is not referenced.

iclustr

(global) INTEGER. Array of size `(2*NPROW*NPCOL)`

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2**i*-1) to *iclustr*(2**i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. *iclustr*(2**k*) ≠ 0 and *iclustr*(2**k*+1) = 0 if and only if *k* is the number of clusters.

iclustr is not referenced if `jobz = 'N'`.

gap

(global)

REAL for pssyevx

DOUBLE PRECISION for pdsyevx.

Array of size NPROW*NPCOL

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*th cluster may be as high as `(C*n)/gap(i)` where *C* is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info*> 0: if (mod(*info*,2)≠0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*=2.0*p?lamch('U').

If (mod(*info*/2,2)≠0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace.The indices of the clusters are stored in the array *iclustr*.

If (mod(*info*/4,2)≠0), then space limit prevented p?syevxf rom computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If (mod(*info*/8,2)≠0), then p?stebz failed to compute eigenvalues. Ensure *abstol*=2.0*p?lamch('U').

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?heev
Computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix.

Syntax
```
call pcheev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, rwork, lrwork, info)
call pzheev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, rwork, lrwork, info)
```

Include Files

Description
The p?heev routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* by calling the recommended sequence of ScaLAPACK routines. The routine assumes a homogeneous system and makes spot checks of the consistency of the eigenvalues across the different processes. A heterogeneous system may return incorrect results without any error messages.

Input Parameters
np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) CHARACTER*1. Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A*(*n*≥ 0).

a (local).

COMPLEX for pcheev

DOUBLE COMPLEX for pzheev.

Block cyclic array of global size (*n*, *n*) and local size (*lld_a*, *LOCc*(*ja* +*n*-1)). On entry, the Hermitian matrix *A*.

If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the Hermitian matrix.

If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the Hermitian matrix.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*. If *desca*(*ctxt_*) is incorrect, p?heev cannot guarantee correct error reporting.

iz, *jz* (global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local).

COMPLEX for pcheev

DOUBLE COMPLEX for pzheev.

Array of size *lwork*.

lwork (local) INTEGER. The size of the array *work*.

If only eigenvalues are requested (*jobz* = 'N'):

lwork≥max(*nb**(*np*0 + 1), 3) + 3**n*

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required:

lwork≥ (*np*0+*nq*0+*nb*)**nb* + 3**n* + n^2

with *nb* = *desca*(*mb_*) = *desca*(*nb_*) = *nb* = *descz*(*mb_*) = *descz*(*nb_*)

*np*0 = numroc(*nn*, *nb*, 0, 0, NPROW).

*nq*0 = numroc(max(*n*, *nb*, 2), *nb*, 0, 0, NPCOL).

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by pxerbla.

rwork

(local).

REAL for pcheev

DOUBLE PRECISION for pzheev.

Workspace array of size *lrwork*.

lrwork

(local) INTEGER. The size of the array *rwork*.

See below for definitions of variables used to define *lrwork*.

If no eigenvectors are requested (*jobz* = 'N'), then *lrwork*≥ 2*n.

If eigenvectors are requested (*jobz* = 'V'), then *lrwork*≥ 2*n + 2*n-2.

If *lrwork* = -1, then *lrwork* is global input and a workspace query is assumed; the routine only calculates the minimum size required for the *rwork* array. The required workspace is returned as the first element of *rwork*, and no error message is issued by pxerbla.

Output Parameters

a

On exit, the lower triangle (if *uplo* = 'L'), or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w

(global).

REAL for pcheev

DOUBLE PRECISION for pzheev.

Array of size *n*. The first *m* elements contain the selected eigenvalues in ascending order.

z

(local).

COMPLEX for pcheev

DOUBLE COMPLEX for pzheev.

Array, global size (*n*, *n*), local size (*lld_z*, *LOCc*(*jz*+*n*-1)).

If *jobz* ='V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

work(1)

On exit, returns adequate workspace to allow optimal performance.

If *jobz* ='N', then *work*(1) = minimal workspace only for eigenvalues.

If *jobz* ='V', then *work*(1) = minimal workspace required to generate all the eigenvectors.

rwork(1) (local)

COMPLEX for pcheev

DOUBLE COMPLEX for pzheev.

On output, *rwork*(1) returns workspace required to guarantee completion.

info (global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*). If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info*> 0:

If *info* = 1 through *n*, the *i*-th eigenvalue did not converge in ?steqr2 after a total of 30**n* iterations.

If *info* = *n*+1, then p?heev detected heterogeneity, and the accuracy of the results cannot be guaranteed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?heevd

Computes all eigenvalues and eigenvectors of a complex Hermitian matrix by using a divide and conquer algorithm.

Syntax

```
call pcheevd(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call pzheevd(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Include Files

Description

The p?heevd routine computes all eigenvalues and eigenvectors of a complex Hermitian matrix *A* by using a divide and conquer algorithm.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) CHARACTER*1. Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* (*n*≥ 0).

a (local).

COMPLEX for pcheevd

DOUBLE COMPLEX for pzheevd.

Block cyclic array of global size (*n*, *n*) and local size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the Hermitian matrix *A*.

If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the Hermitian matrix.

If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the Hermitian matrix.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*. If *desca*(*ctxt_*) is incorrect, p?heevd cannot guarantee correct error reporting.

iz, *jz* (global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local).

COMPLEX for pcheevd

DOUBLE COMPLEX for pzheevd.

Array of size *lwork*.

lwork (local) INTEGER. The size of the array *work*.

If eigenvalues are requested:

lwork = *n* + (*nb*0 + *mq*0 + *nb*)**nb*

with *np*0 = numroc(max(*n*, *nb*, 2), *nb*, 0, 0, NPROW)

*mq*0 = numroc(max(*n*, *nb*, 2), *nb*, 0, 0, NPCOL)

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by pxerbla.

rwork (local).

REAL for pcheevd

DOUBLE PRECISION for pzheevd.

Workspace array of size *lrwork*.

lrwork (local) INTEGER. The size of the array *rwork*.

lrwork≥ 1 + 9*n + 3*np*nq,

with *np* = numroc(*n*, *nb*, *myrow*, *iarow*, NPROW)

nq = numroc(*n*, *nb*, *mycol*, *iacol*, NPCOL)

iwork (local) INTEGER. Workspace array of size *liwork*.

liwork (local) INTEGER , size of *iwork*.

liwork = 7*n + 8*npcol + 2.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L'), or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w (global).

REAL for pcheevd

DOUBLE PRECISION for pzheevd.

Array of size *n*. If *info* = 0, *w* contains the eigenvalues in the ascending order.

z (local).

COMPLEX for pcheevd

DOUBLE COMPLEX for pzheevd.

Array, global size (*n*, *n*), local size (*lld_z*, *LOCc*(*jz*+*n*-1)).

The *z* parameter contains the orthonormal eigenvectors of the matrix *A*.

work(1) On exit, returns adequate workspace to allow optimal performance.

rwork(1) (local)

COMPLEX for pcheevd

DOUBLE COMPLEX for pzheevd.

On output, *rwork*(1) returns workspace required to guarantee completion.

iwork(1) (local).

On return, *iwork*(1) contains the amount of integer workspace required.

info (global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*). If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info*> 0:

If *info* = 1 through *n*, the *i*-th eigenvalue did not converge.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?heevr

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using Relatively Robust Representation.

Syntax

```
call pcheevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, info )
```

```
call pzheevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, info )
```

Include Files

Description

p?heevr computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* distributed in 2D blockcyclic format by calling the recommended sequence of ScaLAPACK routines.

First, the matrix *A* is reduced to complex Hermitian tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in *w*. The eigenvector matrix *Z* is stored in 2D block-cyclic format distributed over all processors.

Input Parameters

jobz

(global) CHARACTER*1

Specifies whether or not to compute the eigenvectors:

= 'N': Compute eigenvalues only.

= 'V': Compute eigenvalues and eigenvectors.

range

(global) CHARACTER*1

= 'A': all eigenvalues will be found.

= 'V': all eigenvalues in the interval [*vl*,*vu*] will be found.

= 'I': the *il*-th through *iu*-th eigenvalues will be found.

uplo (global) CHARACTER*1

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n (global) INTEGER

The number of rows and columns of the matrix *A*. $n \geq 0$

a COMPLEX for pcheevr

COMPLEX*16 for pzheevr

Block-cyclic array, global size (*n*, *n*), local size (*lld_a*, $LOC_c(ja+n-1)$)

Contains the local pieces of the Hermitian distributed matrix *A*. If *uplo* = 'U', only the upper triangular part of *a* is used to define the elements of the Hermitian matrix. If *uplo* = 'L', only the lower triangular part of *a* is used to define the elements of the Hermitian matrix.

ia (global) INTEGER

Global row index in the global matrix *A* that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

ja (global) INTEGER

Global column index in the global matrix *A* that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

desca (global and local) INTEGER array of size *dlen_*. (The ScaLAPACK descriptor length is *dlen_* = 9.)

The array descriptor for the distributed matrix *a*. The descriptor stores details about the 2D block-cyclic storage, see the notes below. If *desca* is incorrect, p?heevr cannot work correctly.

Also note the array alignment requirements specified below

vl REAL for pcheevr

DOUBLE PRECISION for pzheevr

(global)

If *range*='V', the lower bound of the interval to be searched for eigenvalues. Not referenced if *range* = 'A' or 'I'.

vu REAL for pcheevr

DOUBLE PRECISION for pzheevr

(global)

If *range*='V', the upper bound of the interval to be searched for eigenvalues. Not referenced if *range* = 'A' or 'I'.

il (global) INTEGER

If *range*='I', the index (from smallest to largest) of the smallest eigenvalue to be returned. *il*≥ 1.

Not referenced if *range* = 'A'.

iu

(global) INTEGER

If *range*='I', the index (from smallest to largest) of the largest eigenvalue to be returned. min(*il*,*n*) ≤*iu*≤*n*.

Not referenced if *range* = 'A'.

iz

(global) INTEGER

Global row index in the global matrix *Z* that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

jz

(global) INTEGER

Global column index in the global matrix *Z* that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

descz

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*)

work

COMPLEX for pcheevr

COMPLEX*16 for pzheevr

(local workspace) array of size *lwork*

lwork

(local) INTEGER

Size of *work* array, must be at least 3.

If only eigenvalues are requested:

lwork≥*n* + max(*nb* * (*np00* + 1), *nb* * 3)

If eigenvectors are requested:

lwork≥*n* + (*np00* + *mq00* + *nb*) * *nb*

For definitions of *np00* and *mq00*, see *lrwork*.

For optimal performance, greater workspace is needed, i.e.

lwork≥ max(*lwork*, *nhetrd_lwork*)

Where *lwork* is as defined above, and

nhetrd_lwork = *n* + 2*(*anb*+1)*(4**nps*+2) + (*nps* + 1) * *nps*

ictxt = *desca*(*ctxt_*)

anb = pjlaenv(*ictxt*, 3, 'PCHETTRD', 'L', 0, 0, 0, 0)

sqnpc = sqrt(real(*nprow* * *npcol*))

nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

rwork

REAL for pcheevr

DOUBLE PRECISION for pzheevr

(local workspace) array of size *lrwork*

lrwork

(local) INTEGER

Size of *rwork*, must be at least 3.

See below for definitions of variables used to define *lrwork*.

If no eigenvectors are requested (*jobz* = 'N') then

lrwork≥ 2 + 5 * *n* + max(12 * *n*, *nb* * (*np00* + 1))

If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required is:

lrwork≥ 2 + 5 * *n* + max(18**n*, *np00* * *mq00* + 2 * *nb* * *nb*) +

(2 + iceil(*neig*, *nprow***npcol*))**n*

NOTE
iceil(*x*, *y*) is the ceiling of *x*/*y*.

Variable definitions:

neig = number of eigenvectors requested

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*)

nn = max(*n*, *nb*, 2)

desca(*rsrc_*) = *desca*(*csrc_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0

np00 = numroc(*nn*, *nb*, 0, 0, *nprow*)

mq00 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, *npcol*)

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*), and *nprow* and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lrwork* = -1, then *lrwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla

iwork

(local workspace) INTEGER array of size *liwork*

liwork

(local) INTEGER

size of *iwork*

Let *nnp* = max(*n*, *nprow***npcol* + 1, 4). Then:

liwork≥ 12**nnp* + 2**n* when the eigenvectors are desired

liwork≥ 10**nnp* + 2**n* when only the eigenvalues have to be computed

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla

OUTPUT Parameters

a

The lower triangle (if *uplo*='L') or the upper triangle (if *uplo*='U') of *a*, including the diagonal, is destroyed.

m

(global) INTEGER

Total number of eigenvalues found. 0 ≤*m*≤*n*.

nz

(global) INTEGER

Total number of eigenvectors computed. 0 ≤*nz*≤*m*.

The number of columns of *z* that are filled.

If *jobz*≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m*

w

REAL for pcheevr

DOUBLE PRECISION for pzheevr

(global) array of size *n*

On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z

COMPLEX for pcheevr

COMPLEX*16 for pzheevr

(local) array, global size (*n*, *n*), local size (*lld_z*, LOC_c(*jz*+*n*-1))

If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues.

If *jobz* = 'N', then *z* is not referenced.

work

work(1) returns workspace adequate workspace to allow optimal performance.

rwork

On return, *rwork*(1) contains the optimal amount of workspace required for efficient execution. if *jobz*='N' *rwork*(1) = optimal amount of workspace required to compute the eigenvalues. if *jobz*='V' *rwork*(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors.

iwork

On return, *iwork*(1) contains the amount of integer workspace required.

info

(global) INTEGER

= 0: successful exit

< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The distributed submatrices *a*(*ia*:*, *ja*:*) and *z*(*iz*:*iz*+*m*-1,*jz*:*jz*+*n*-1) must satisfy the following alignment properties:

1. Identical (quadratic) dimension: *desca*(*m_*) = *descz*(*m_*) = *desca*(*n_*) = *descz*(*n_*)
2. Quadratic conformal blocking: *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*), *desca*(*rsrc_*) = *descz*(*rsrc_*)
3. mod(*ia*-1, *mb_a*) = mod(*iz*-1, *mb_z*) = 0

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call pcheevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr, gap, info)
```

```
call pzheevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr, gap, info)
```

Include Files

Description

The p?heevx routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) CHARACTER*1. Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range (global) CHARACTER*1. Must be 'A', 'V', or 'I'.

If *range* = 'A', all eigenvalues will be found.

If *range* = 'V', all eigenvalues in the half-open interval [*vl*, *vu*] will be found.

If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* (*n* ≥ 0).

a (local).

COMPLEX for pcheevx

DOUBLE COMPLEX for pzheevx.

Block cyclic array of global size (*n*, *n*) and local size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the Hermitian matrix *A*.

If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the Hermitian matrix.

If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the Hermitian matrix.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*. If *desca*(*ctxt_*) is incorrect, p?heevx cannot guarantee correct error reporting.

vl, *vu* (global)

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; not referenced if *range* = 'A' or 'I'.

il, *iu* (global)

INTEGER. If *range* ='I', the indices of the smallest and largest eigenvalues to be returned.

Constraints:

il ≥ 1; min(*il*,*n*) ≤ *iu* ≤ *n*.

Not referenced if *range* = 'A' or 'V'.

abstol (global).

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*, *b*] of width less than or equal to *abstol*+*eps**max(|*a*|,|*b*|), where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S'), not zero. If this routine returns with ((mod(*info*,2)≠0).*or*.(mod(*info*/8,2)≠0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

orfac (global). REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative.

orfac should be identical on all processes.

iz, *jz* (global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local).

COMPLEX for pcheevx

DOUBLE COMPLEX for pzheevx.

Array of size *lwork*.

lwork (local) INTEGER. The size of the array *work*.

If only eigenvalues are requested:

lwork≥*n* + max(*nb**(*np*0 + 1), 3)

If eigenvectors are requested:

lwork≥*n* + (*np*0+*mq*0+*nb*)***nb*

with *nq*0 = numroc(*nn*, *nb*, 0, 0, NPCOL).

lwork≥ 5**n* + max(5**nn*, *np*0**mq*0+2**nb***nb*) + iceil(*neig*, NPROW*NPCOL)**nn*

For optimal performance, greater workspace is needed, that is

lwork≥max(*lwork*, *nhetrd_lwork*)

where *lwork* is as defined above, and *nhetrd_lwork* = *n* + 2*(*anb* +1)*(4**nps*+2) + (*nps*+1)**nps*

ictxt = *desca*(*ctxt_*)

anb = pjlaenv(*ictxt*, 3, 'pchettrd', 'L', 0, 0, 0, 0)

sqnpc = *sqrt*(*dble*(NPROW * NPCOL))

nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

rwork (local)

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Workspace array of size *lrwork*.

lrwork (local) INTEGER. The size of the array *work*.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then *lrwork*≥ 5**nn*+4**n*.

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

lrwork≥ 4**n* + max(5**nn*, *np*0**mq*0+2**nb***nb*) + iceil(*neig*, NPROW*NPCOL)**nn*

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following values to *lrwork*:

(*clustersize*-1)**n*,

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

{*w*(*k*),..., *w*(*k*+*clustersize*-1)|*w*(*j*+1) ≤ *w*(*j*) +*orfac**2*norm(*A*)}.

Variable definitions:

neig = number of eigenvectors requested;

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);

nn = max(*n*, NB, 2);

desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0;

*np*0 = numroc(*nn*, *nb*, 0, 0, NPROW);

*mq*0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL);

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

When *lrwork* is too small:

If *lwork* is too small to guarantee orthogonality, p?heevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues. If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned. Note that when *range*='V', p?heevx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow p?heevx to compute the eigenvalues, p?heevx will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality and performance:

If *clustersize* ≥ *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1)p?stein will perform no better than ?stein on 1 processor.

For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize*>*n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, size of *iwork*.

liwork ≥ 6**nnp*

Where: *nnp* = max(*n*, NPROW*NPCOL+1, 4)

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L'), or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m (global) INTEGER. The total number of eigenvalues found; $0 \le m \le n$.

nz (global) INTEGER. Total number of eigenvectors computed. $0 \le nz \le m$.

The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p?heevx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*≤*descz*(*n_*)) and sufficient workspace to compute them. (See *lwork*). p?heevx is always able to detect insufficient space without computation unless *range*='V'.

w (global).

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Array of size *n*. The first *m* elements contain the selected eigenvalues in ascending order.

z (local).

COMPLEX for pcheevx

DOUBLE COMPLEX for pzheevx.

Array, global size (*n*, *n*), local size (*lld_z*, *LOCc*(*jz*+*n*-1)).

If *jobz* ='V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

work(1) On exit, returns adequate workspace to allow optimal performance.

rwork (local).

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Array of size *lrwork*. On return, *rwork*(1) contains the optimal amount of workspace required for efficient execution.

If *jobz*='N'*rwork*(1) = optimal amount of workspace required to compute eigenvalues efficiently.

If *jobz*='V'*rwork*(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If *range*='V', it is assumed that all eigenvectors may be required.

iwork(1) (local)

On return, *iwork*(1) contains the amount of integer workspace required.

ifail (global) INTEGER.

Array of size *n*.

If *jobz* ='V', then on normal exit, the first *m* elements of *ifail* are zero. If (mod(*info*,2)≠0) on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

iclustr

(global) INTEGER.

Array of size 2*NPROW*NPCOL.

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i*-1) to *iclustr*(2*i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. (*iclustr*(2**k*)≠0 and *iclustr*(2**k*+1)=0) if and only if *k* is the number of clusters. *iclustr* is not referenced if *jobz* = 'N'.

gap

(global)

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Array of size (NPROW*NPCOL)

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as (*C***n*)/ *gap*(*i*) where *C* is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*). If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info*> 0:

If (mod(*info*,2)≠0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*=2.0*p?lamch('U')

If (mod(*info*/2,2)≠0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace.The indices of the clusters are stored in the array *iclustr*.

If (mod(*info*/4,2)≠0), then space limit prevented p?syevx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If (mod(*info*/8,2)≠0), then p?stebz failed to compute eigenvalues. Ensure *abstol*=2.0*p?lamch('U').

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gesvd

Computes the singular value decomposition of a general matrix, optionally computing the left and/or right singular vectors.

Syntax

```
call psgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt, descvt, work, lwork, info)
call pdgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt, descvt, work, lwork, info)
call pcgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt, descvt, work, lwork, rwork, info)
call pzgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt, descvt, work, lwork, rwork, info)
```

Include Files

Description

The p?gesvd routine computes the singular value decomposition (SVD) of an *m*-by-*n* matrix *A*, optionally computing the left and/or right singular vectors. The SVD is written

$A = U*\Sigma*V^T$,

where Σ is an *m*-by-*n* matrix that is zero except for its min(*m*, *n*) diagonal elements, *U* is an *m*-by-*m* orthogonal matrix, and *V* is an *n*-by-*n* orthogonal matrix. The diagonal elements of Σ are the singular values of *A* and the columns of *U* and *V* are the corresponding right and left singular vectors, respectively. The singular values are returned in array *s* in decreasing order and only the first min(*m*,*n*) columns of *U* and rows of $vt = V^T$ are computed.

NOTE
The distributed submatrix sub(*A*) must verify certain alignment properties. These expressions must be true:

- *mb_a* = *nb_a* = *nb*
- *iroffa* = *icoffa*

where:

- *iroffa* = mod(*ia*-1, *nb*)
- *icoffa* = mod(*ja*-1, *nb*)

Input Parameters

mp = number of local rows in *A* and *U*

nq = number of local columns in *A* and *VT*

size = min(*m*, *n*)

sizeq = number of local columns in *U*

sizep = number of local rows in *VT*

jobu	(global) CHARACTER*1. Specifies options for computing all or part of the matrix *U*. If *jobu* = 'V', the first *size* columns of *U* (the left singular vectors) are returned in the array *u*; If *jobu* ='N', no columns of *U* (no left singular vectors)are computed.
jobvt	(global) CHARACTER*1. Specifies options for computing all or part of the matrix V^T. If *jobvt* = 'V', the first *size* rows of V^T (the right singular vectors) are returned in the array *vt*; If *jobvt* = 'N', no rows of V^T(no right singular vectors) are computed.
m	(global) INTEGER. The number of rows of the matrix *A* (*m*≥ 0).
n	(global) INTEGER. The number of columns in *A* (*n*≥ 0).
a	(local). REAL for psgesvd DOUBLE PRECISION for pdgesvd COMPLEX for pcgesvd COMPLEX*16 for pzgesvd Block cyclic array, global size (*m*, *n*), local size (*mp*, *nq*).
ia, *ja*	(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.
iu, *ju*	(global) INTEGER. The row and column indices in the global matrix *U* indicating the first row and the first column of the submatrix *U*, respectively.
descu	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *U*.
ivt, *jvt*	(global) INTEGER. The row and column indices in the global matrix *VT* indicating the first row and the first column of the submatrix *VT*, respectively.
descvt	(global and local)INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *VT*.
work	(local).REAL for psgesvd

DOUBLE PRECISION for pdgesvd

COMPLEX for pcgesvd

COMPLEX*16 for pzgesvd

Workspace array of size *lwork*

lwork (local) INTEGER. The size of the array *work*;

lwork > 2 + 6*sizeb + max(*watobd*, *wbdtosvd*),

where *sizeb* = max(*m*, *n*), and *watobd* and *wbdtosvd* refer, respectively, to the workspace required to bidiagonalize the matrix *A* and to go from the bidiagonal matrix to the singular value decomposition *USVT*.

For *watobd*, the following holds:

watobd = max(max(*wp?lange*,*wp?gebrd*), max(*wp?lared2d*, *wp?lared1d*)),

where *wp?lange*, *wp?lared1d*, *wp?lared2d*, *wp?gebrd* are the workspaces required respectively for the subprograms p?lange, p?lared1d, p?lared2d, p?gebrd. Using the standard notation

mp = numroc(*m*, *mb*, MYROW, *desca*(*ctxt_*), NPROW),

nq = numroc(*n*, *nb*, MYCOL, *desca*(*lld_*), NPCOL),

the workspaces required for the above subprograms are

wp?lange = *mp*,

wp?lared1d = *nq*0,

wp?lared2d = *mp*0,

wp?gebrd = *nb**(*mp* + *nq* + 1) + *nq*,

where *nq*0 and *mp*0 refer, respectively, to the values obtained at MYCOL = 0 and MYROW = 0. In general, the upper limit for the workspace is given by a workspace required on processor (0,0):

watobd ≤ *nb**(*mp*0 + *nq*0 + 1) + *nq*0.

In case of a homogeneous process grid this upper limit can be used as an estimate of the minimum workspace for every processor.

For *wbdtosvd*, the following holds:

wbdtosvd = *size**(*wantu***nru* + *wantvt***ncvt*) + max(*w?bdsqr*, max(*wantu***wp?ormbrqln*, *wantvt***wp?ormbrprt*)),

where

wantu(*wantvt*) = 1, if left/right singular vectors are wanted, and *wantu*(*wantvt*) = 0, otherwise. *w?bdsqr*, *wp?ormbrqln*, and *wp?ormbrprt* refer respectively to the workspace required for the subprograms ?bdsqr, p?ormbr(*qln*), and p?ormbr(*prt*), where *qln* and *prt* are the values of the arguments *vect*, *side*, and *trans* in the call to p?ormbr. *nru* is equal to the local number of rows of the matrix *U* when distributed 1-dimensional "column" of processes. Analogously, *ncvt* is equal to the local number of columns of the matrix *VT* when distributed across 1-dimensional "row" of processes. Calling the LAPACK procedure ?bdsqr requires

w?bdsqr = max(1, 2**size* + (2**size* - 4)* max(*wantu*, *wantvt*))

on every processor. Finally,

wp?ormbrqln = max((*nb**(*nb*-1))/2, (*sizeq*+*mp*)**nb*)+*nb***nb*,

wp?ormbrprt = max((*mb**(*mb*-1))/2, (*sizep*+*nq*)**mb*)+*mb***mb*,

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum size for the work array. The required workspace is returned as the first element of *work* and no error message is issued by pxerbla.

rwork

REAL for pcgesvd

DOUBLE PRECISION for pzgesvd

Workspace array of size 1 + 4**sizeb*. Not used for psgesvd and pdgesvd.

Output Parameters

a

On exit, the contents of *a* are destroyed.

s

(global). REAL for psgesvd and pcgesvd

DOUBLE PRECISION for pdgesvd and pzgesvd

Array of size *size*.

Contains the singular values of *A* sorted so that *s*(i) ≥*s*(i+1).

u

(local). REAL for psgesvd

DOUBLE PRECISION for pdgesvd

COMPLEX for pcgesvd

COMPLEX*16 for pzgesvd

local size (*mp*, *sizeq*), global size (*m*, *size*)

If *jobu* = 'V', *u* contains the first min(*m*, *n*) columns of *U*.

If *jobu* = 'N' or 'O', *u* is not referenced.

vt

(local). REAL for psgesvd

DOUBLE PRECISION for pdgesvd

COMPLEX for pcgesvd

COMPLEX*16 for pzgesvd

local size (*sizep*, *nq*), global size (*size*, *n*)

If *jobvt* = 'V', *vt* contains the first *size* rows of V^Tif *jobu* = 'N', *vt* is not referenced.

work

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork

On exit, if *info* = 0, then *rwork*(1) returns the required size of *rwork*.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0, If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* > 0 *i*, then if ?bdsqr did not converge,

If *info* = min(*m*,*n*) + 1, then p?gesvd has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?gesvd cannot be guaranteed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

call pssygvx(*ibtype*, *jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

call pdsygvx(*ibtype*, *jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

Include Files

Description

The p?sygvxroutine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

sub(*A*)**x* = λ*sub(*B*)**x*, sub(*A*) sub(*B*)**x* = λ**x*, or sub(*B*)*sub(*A*)**x* = λ**x*.

Here *x* denotes eigen vectors, λ (*lambda*) denotes eigenvalues, sub(*A*) denoting A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) is assumed to symmetric, and sub(*B*) denoting *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1) is also positive definite.

Input Parameters

ibtype

(global) INTEGER. Must be 1 or 2 or 3.

Specifies the problem type to be solved:

If *ibtype* = 1, the problem type is sub(*A*)**x* = *lambda**sub(*B*)**x*;

If *ibtype* = 2, the problem type is sub(*A*)*sub(*B*)**x* = *lambda***x*;

If *ibtype* = 3, the problem type is sub(*B*)*sub(*A*)**x* = *lambda***x*.

jobz

(global) CHARACTER*1. Must be 'N' or 'V'.

If *jobz* ='N', then compute eigenvalues only.

If *jobz* ='V', then compute eigenvalues and eigenvectors.

range (global) CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues in the interval: [*vl*, *vu*]

If *range* = 'I', the routine computes eigenvalues with indices *il* through *iu*.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *a* and *b* store the upper triangles of sub(*A*) and sub (*B*);

If *uplo* = 'L', arrays *a* and *b* store the lower triangles of sub(*A*) and sub (*B*).

n (global) INTEGER. The order of the matrices sub(*A*) and sub (*B*), $n \geq 0$.

a (local)

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*. If *desca*(*ctxt_*) is incorrect, p?sygvx cannot guarantee correct error reporting.

b (local). REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

Pointer into the local memory to an array of size (*lld_b*, *LOCc*(*jb*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*B*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*B*) contains the upper triangular part of the matrix.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ib, *jb* (global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*. *descb*(*ctxt_*) must be equal to *desca*(*ctxt_*).

vl, *vu* (global)

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* (global)

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1$, $\min(il, n) \leq iu \leq n$

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol (global)

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to

abstol + *eps**max(|*a*|,|*b*|),

where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S') not zero. If this routine returns with ((mod(*info*,2)≠0) or (*mod*(*info*/8,2)≠0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

orfac (global).

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be

reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, *jz*

(global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work

(local)

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

Workspace array of size *lwork*

lwork

(local) INTEGER.

Size of the array *work*. See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then *lwork* ≥ 5**n* + max(5**nn*, NB*(*np*0 + 1)).

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

lwork ≥ 5**n* + max(5**nn*, *np*0**mq*0 + 2**nb***nb*) + iceil(*neig*, NPROW*NPCOL)**nn*.

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality at the cost of potentially poor performance you should add the following to *lwork*:

(*clustersize*-1)**n*,

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

{*w*(*k*),..., *w*(*k*+*clustersize*-1)|*w*(*j*+1) ≤ *w*(*j*) + *orfac**2*norm*(*A*)}

Variable definitions:

neig = number of eigenvectors requested,

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*),

nn = max(*n*, *nb*, 2),

desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0,

*np*0 = numroc(*nn*, *nb*, 0, 0, NPROW),

*mq*0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL)

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

If *lwork* is too small to guarantee orthogonality, p?syevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, p?sygvx computes the eigenvalues and as many eigenvectors as possible.

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

lwork ≥ max(*lwork*, 5*n* + *nsytrd_lwopt*, *nsygst_lwopt*), where

lwork, as defined previously, depends upon the number of eigenvectors requested, and

nsytrd_lwopt = *n* + 2*(*anb*+1)*(4**nps*+2) + (*nps*+3)**nps*

nsygst_lwopt = 2**np*0**nb* + *nq*0**nb* + *nb***nb*

anb = pjlaenv(*desca*(*ctxt_*), 3, p?syttrd ', 'L', 0, 0, 0, 0)

sqnpc = int(*sqrt*(*dble*(NPROW * NPCOL)))

nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)

NB = *desca*(*mb_*)

*np*0 = numroc(*n*, *nb*, 0, 0, NPROW)

*nq*0 = numroc(*n*, *nb*, 0, 0, NPCOL)

numroc is a ScaLAPACK tool functions;

pjlaenv is a ScaLAPACK environmental inquiry function

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

For large n, no extra workspace is needed, however the biggest boost in performance comes for small n, so it is wise to provide the extra workspace (typically less than a Megabyte per process).

If *clustersize* ≥ *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, *clustersize* = *n*-1) p?stein will perform no better than ?stein on a single processor.

For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize*>*n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER , size of *iwork*.

liwork ≥ 6*nnp

Where:

nnp = max(*n*, NPROW*NPCOL + 1, 4)

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit,

If *jobz* = 'V', and if *info* = 0, sub(*A*) contains the distributed matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

for *ibtype* = 1 or 2, Z^T*sub(*B*)**Z* = *i*;

for *ibtype* = 3, Z^T*inv(sub(*B*))**Z* = *i*.

If *jobz* = 'N', then on exit the upper triangle (if *uplo*='U') or the lower triangle (if *uplo*='L') of sub(*A*), including the diagonal, is destroyed.

b On exit, if *info* ≤ *n*, the part of sub(*B*) containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization sub(*B*) = U^T**U* or sub(*B*) = *L***L*^{T}.

m (global) INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*.

nz (global) INTEGER.

Total number of eigenvectors computed. 0 ≤ *nz* ≤ *m*. The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p?sygvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*≤*descz*(*n_*)) and sufficient workspace to compute them. (See *lwork* below.) p?sygvx is always able to detect insufficient space without computation unless *range*='V'.

w (global)

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

Array of size *n*. On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z (local).

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

global size (*n*, *n*), local size `(lld_z, LOCc(jz+n-1))`.

If `jobz = 'V'`, then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If `jobz = 'N'`, then *z* is not referenced.

work

If *jobz*=`'N'` *work*`(1)` = optimal amount of workspace required to compute eigenvalues efficiently

If `jobz = 'V'` *work*`(1)` = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If *range*=`'V'`, it is assumed that all eigenvectors may be required.

ifail

(global) `INTEGER`.

Array of size *n*.

ifail provides additional information when *info*≠0

If `(mod(info/16,2)≠0)` then *ifail*`(1)` indicates the order of the smallest minor which is not positive definite. If `(mod(info,2)≠0)` on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.

If neither of the above error conditions hold and `jobz = 'V'`, then the first *m* elements of *ifail* are set to zero.

iclustr

(global) `INTEGER`.

Array of size `(2*NPROW*NPCOL)`. This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*`(2*i-1)` to *iclustr*`(2*i)`, could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*`()` is a zero terminated array.

`(iclustr(2*k)≠0.and. iclustr(2*k+1)=0)` if and only if *k* is the number of clusters *iclustr* is not referenced if `jobz = 'N'`.

gap

(global)

`REAL` for `pssygvx`

`DOUBLE PRECISION` for `pdsygvx`.

Array of size `NPROW*NPCOL`. This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as `(C*n)/gap(i)`, where *C* is a small constant.

info

(global) `INTEGER`.

If `info = 0`, the execution is successful.

If *info* <0: the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info*> 0:

If (mod(*info*,2)≠0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If (mod(*info*,2,2)≠0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If (mod(*info*/4,2)≠0), then space limit prevented p?sygvx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If (mod(*info*/8,2)≠0), then p?stebz failed to compute eigenvalues.

If (mod(*info*/16,2)≠0), then *B* was not positive definite. *ifail(1)* indicates the order of the smallest minor which is not positive definite.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

call pchegvx(*ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr, gap, info*)

call pzhegvx(*ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr, gap, info*)

Include Files

Description

The p?hegvx routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

sub(*A*)**x* = λ*sub(*B*)**x*, sub(*A*)*sub(*B*)**x* = λ**x*, or sub(*B*)*sub(*A*)**x* = λ**x*.

Here sub (*A*) denoting *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(*B*) are assumed to be Hermitian and sub(*B*) denoting *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1) is also positive definite.

Input Parameters

ibtype

(global) INTEGER. Must be 1 or 2 or 3.

Specifies the problem type to be solved:

If *ibtype* = 1, the problem type is

sub(*A*)**x* = *lambda**sub(*B*)**x*;

If *ibtype* = 2, the problem type is

sub(*A*)*sub(*B*)**x* = *lambda***x*;

If *ibtype* = 3, the problem type is

sub(*B*)*sub(*A*)**x* = *lambda***x*.

jobz

(global) CHARACTER*1. Must be 'N' or 'V'.

If *jobz* ='N', then compute eigenvalues only.

If *jobz* ='V', then compute eigenvalues and eigenvectors.

range

(global) CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues in the interval: [*vl*, *vu*]

If *range* = 'I', the routine computes eigenvalues with indices *il* through *iu*.

uplo

(global) CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *a* and *b* store the upper triangles of sub(*A*) and sub (*B*);

If *uplo* = 'L', arrays *a* and *b* store the lower triangles of sub(*A*) and sub (*B*).

n

(global) INTEGER.

The order of the matrices sub(*A*) and sub (*B*) (*n*≥ 0).

a

(local)

COMPLEX for pchegvx

DOUBLE COMPLEX for pzhegvx.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ia, *ja*

(global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *A*. If *desca*(*ctxt_*) is incorrect, p?hegvx cannot guarantee correct error reporting.

b (local).

COMPLEX for pchegvx

DOUBLE COMPLEX for pzhegvx.

Pointer into the local memory to an array of size (*lld_b*, *LOCc*(*jb*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(*B*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*B*) contains the upper triangular part of the matrix.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*B*) contains the lower triangular part of the matrix.

ib, *jb* (global) INTEGER.

The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *B*. *descb*(*ctxt_*) must be equal to *desca*(*ctxt_*).

vl, *vu* (global)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* (global)

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1$, $\min(il, n) \leq iu \leq n$

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol (global)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to

abstol + *eps**max(|*a*|,|*b*|),

where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S') not zero. If this routine returns with ((mod(*info*,2)≠0).*or*. * (*mod*(*info*/8,2)≠0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

NOTE
mod(*x*,*y*) is the integer remainder of *x*/*y*.

orfac (global).

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0E-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, *jz* (global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*.*descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local)

COMPLEX for pchegvx

DOUBLE COMPLEX for pzhegvx.

Workspace array of size *lwork*

lwork (local).

INTEGER. The size of the array *work*.

If only eigenvalues are requested:

lwork ≥ *n*+ max(NB*(*np*0 + 1), 3)

If eigenvectors are requested:

lwork ≥ *n* + (*np*0+ *mq*0 + NB)*NB

with *nq*0 = numroc(*nn*, NB, 0, 0, NPCOL).

For optimal performance, greater workspace is needed, that is

lwork ≥ max(*lwork*, *n*, *nhetrd_lwopt*, *nhegst_lwopt*)

where *lwork* is as defined above, and

nhetrd_lwork = 2*(*anb*+1)*(4**nps*+2) + (*nps* + 1)**nps*;

nhegst_lwopt = 2**np*0**nb* + *nq*0**nb* + *nb***nb*

nb = *desca*(*mb_*)

*np*0 = numroc(*n*, *nb*, 0, 0, NPROW)

*nq*0 = numroc(*n*, *nb*, 0, 0, NPCOL)

ictxt = *desca*(*ctxt_*)

anb = pjlaenv(*ictxt*, 3, 'p?hettrd', 'L', 0, 0, 0, 0)

sqnpc = *sqrt*(*dble*(NPROW * NPCOL))

nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)

numroc is a ScaLAPACK tool functions;

pjlaenv is a ScaLAPACK environmental inquiry function MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

rwork (local)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

Workspace array of size *lrwork*.

lrwork (local) INTEGER. The size of the array *rwork*.

See below for definitions of variables used to define *lrwork*.

If no eigenvectors are requested (*jobz* = 'N'), then *lrwork* ≥ 5**nn*+4**n*

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

lrwork ≥ 4**n* + max(5**nn*, *np*0**mq*0)+iceil(*neig*, NPROW*NPCOL)**nn*

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following value to *lrwork*:

(*clustersize*-1)**n*,

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

{*w*(*k*),..., *w*(*k*+*clustersize*-1)|*w*(*j*+1) ≤ *w*(*j*)+*orfac**2*norm*(*A*)}

Variable definitions:

neig = number of eigenvectors requested;

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);

nn = max(*n*, *nb*, 2);

desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0 ;

*np*0 = numroc(*nn*, *nb*, 0, 0, NPROW);

*mq*0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL);

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*).

When *lrwork* is too small:

If *lwork* is too small to guarantee orthogonality, p?hegvx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -25 is returned. Note that when *range*='V', p?hegvx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow p?hegvx to compute the eigenvalues, p?hegvx will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:

If *clustersize* > *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) p?stein will perform no better than ?stein on 1 processor.

For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize*>*n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lrwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER , size of *iwork*.

liwork ≥ 6**nnp*

Where: *nnp* = max(*n*, NPROW*NPCOL + 1, 4)

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, if *jobz* = 'V', then if *info* = 0, sub(*A*) contains the distributed matrix *Z* of eigenvectors.

The eigenvectors are normalized as follows:

If *ibtype* = 1 or 2, then Z^H*sub(*B*)**Z* = *i*;

If *ibtype* = 3, then Z^H*inv(sub(*B*))**Z* = *i*.

If *jobz* = 'N', then on exit the upper triangle (if *uplo*='U') or the lower triangle (if *uplo*='L') of sub(*A*), including the diagonal, is destroyed.

b — On exit, if *info* ≤ *n*, the part of sub(*B*) containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization sub(*B*) = U^H**U*, or sub(*B*) = *L*L^H.

m — (global) INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*.

nz — (global) INTEGER. Total number of eigenvectors computed. 0 < *nz* < *m*. The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p?hegvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*≤*descz*(*n_*)) and sufficient workspace to compute them. (See *lwork* below.) The routine p?hegvx is always able to detect insufficient space without computation unless *range* = 'V'.

w — (global)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

Array of size *n*. On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z — (local).

COMPLEX for pchegvx

DOUBLE COMPLEX for pzhegvx.

global size (*n*, *n*), local size (*lld_z*, *LOCc*(*jz*+*n*-1)).

If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

work — On exit, *work*(1) returns the optimal amount of workspace.

rwork — On exit, *rwork*(1) contains the amount of workspace required for optimal efficiency

If *jobz*='N'*rwork*(1) = optimal amount of workspace required to compute eigenvalues efficiently

If *jobz*='V'*rwork*(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If *range*='V', it is assumed that all eigenvectors may be required when computing optimal workspace.

ifail

(global) INTEGER.

Array of size *n*.

ifail provides additional information when *info*≠0

If (*mod*(*info*/16,2)≠0), then *ifail*(1) indicates the order of the smallest minor which is not positive definite.

If (*mod*(*info*,2)≠0) on exit, then *ifail*(1) contains the indices of the eigenvectors that failed to converge.

If neither of the above error conditions are held, and *jobz* = 'V', then the first *m* elements of *ifail* are set to zero.

iclustr

(global) INTEGER.

Array of size (2*NPROW*NPCOL). This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i*–1) to *iclustr*(2*i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal.

iclustr() is a zero terminated array. (*iclustr*(2*k*) ≠0.*and*.*clustr*(2*k*+1)=0) if and only if *k* is the number of clusters.

iclustr is not referenced if *jobz* = 'N'.

gap

(global)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

Array of size NPROW*NPCOL.

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as (*C**n*)/ *gap*(*i*), where *C* is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* <0: the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = –(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = –*i*.

If *info*> 0:

If (mod(*info*,2)≠0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If (mod(*info*,2,2)≠0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If (mod(*info*/4,2)≠0), then space limit prevented p?sygvx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If (mod(*info*/8,2)≠0), then p?stebz failed to compute eigenvalues.

If (mod(*info*/16,2)≠0), then *B* was not positive definite. *ifail(1)* indicates the order of the smallest minor which is not positive definite.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

ScaLAPACK Auxiliary Routines

ScaLAPACK Auxiliary Routines

Routine Name	Data Types	Description
b?laapp	s,d	Multiplies a matrix with an orthogonal matrix.
b?laexc	s,d	Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
b?trexc	s,d	Reorders the Schur factorization of a general matrix.
p?lacgv	c,z	Conjugates a complex vector.
p?max1	c,z	Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS p?amax, but using the absolute value to the real part).
pmpcol	s,d	*Finds the collaborators of a process.*
pmpim2	s,d	Computes the eigenpair range assignments for all processes.
?combamax1	c,z	Finds the element with maximum real part absolute value and its corresponding global index.
p?sum1	sc,dz	Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum, but using the true absolute value.
p?dbtrsv	s,d,c,z	Computes an *LU* factorization of a general tridiagonal matrix with no pivoting. The routine is called by p?dbtrs.
p?dttrsv	s,d,c,z	Computes an *LU* factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by p?dttrs.
p?gebal	s,d	Balances a general real/complex matrix.
p?gebd2	s,d,c,z	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).
p?gehd2	s,d,c,z	Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).
p?gelq2	s,d,c,z	Computes an *LQ* factorization of a general rectangular matrix (unblocked algorithm).

Routine Name	Data Types	Description
p?geql2	s,d,c,z	Computes a *QL* factorization of a general rectangular matrix (unblocked algorithm).
p?geqr2	s,d,c,z	Computes a *QR* factorization of a general rectangular matrix (unblocked algorithm).
p?gerq2	s,d,c,z	Computes an *RQ* factorization of a general rectangular matrix (unblocked algorithm).
p?getf2	s,d,c,z	Computes an *LU* factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).
p?labrd	s,d,c,z	Reduces the first *nb* rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.
p?lacon	s,d,c,z	Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.
p?laconsb	s,d	Looks for two consecutive small subdiagonal elements.
p?lacp2	s,d,c,z	Copies all or part of a distributed matrix to another distributed matrix.
p?lacp3	s,d	Copies from a global parallel array into a local replicated array or vice versa.
p?lacpy	s,d,c,z	Copies all or part of one two-dimensional array to another.
p?laevswp	s,d,c,z	Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.
p?lahrd	s,d,c,z	Reduces the first *nb* columns of a general rectangular matrix A so that elements below the k^{th} subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.
p?laiect	s,d,c,z	Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).
p?lamve	s, d	Copies all or part of one two-dimensional distributed array to another.
p?lange	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.
p?lanhs	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.
p?lansy, p?lanhe	s,d,c,z/c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a real symmetric or complex Hermitian matrix.
p?lantr	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.

Routine Name	Data Types	Description
p?lapiv	s,d,c,z	Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.
p?laqge	s,d,c,z	Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ.
p?laqr0	s,d	Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.
p?laqr1	s,d	Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.
p?laqr2	s,d	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
p?laqr3	s,d	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
p?laqr4	s,d	Computes the eigenvalues of a Hessenberg matrix, and optionally computes the matrices from the Schur decomposition.
p?laqr5	s,d	Performs a single small-bulge multi-shift QR sweep.
p?laqsy	s,d,c,z	Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ.
p?lared1d	s,d	Redistributes an array assuming that the input array *bycol* is distributed across rows and that all process columns contain the same copy of *bycol*.
p?lared2d	s,d	Redistributes an array assuming that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow* .
p?larf	s,d,c,z	Applies an elementary reflector to a general rectangular matrix.
p?larfb	s,d,c,z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
p?larfc	c,z	Applies the conjugate transpose of an elementary reflector to a general matrix.
p?larfg	s,d,c,z	Generates an elementary reflector (Householder matrix).
p?larft	s,d,c,z	Forms the triangular vector *T* of a block reflector $H=I-VTV^H$
p?larz	s,d,c,z	Applies an elementary reflector as returned by p?tzrzf to a general matrix.
p?larzb	s,d,c,z	Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzrzf to a general matrix.
p?larzc	c,z	Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzrzf to a general matrix.

Routine Name	Data Types	Description
p?larzt	s,d,c,z	Forms the triangular factor *T* of a block reflector $H=I-VTV^H$ as returned by p?tzrzf.
p?lascl	s,d,c,z	Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from}.
p?laset	s,d,c,z	Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β.
p?lasmsub	s,d	Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.
p?lassq	s,d,c,z	Updates a sum of squares represented in scaled form.
p?laswp	s,d,c,z	Performs a series of row interchanges on a general rectangular matrix.
p?latra	s,d,c,z	Computes the trace of a general square distributed matrix.
p?latrd	s,d,c,z	Reduces the first *nb* rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
p?latrz	s,d,c,z	Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.
p?lauu2	s,d,c,z	Computes the product UU^H or L^HL, where *U* and *L* are upper or lower triangular matrices (local unblocked algorithm).
p?lauum	s,d,c,z	Computes the product UU^H or L^HL, where *U* and *L* are upper or lower triangular matrices.
p?lawil	s,d	Forms the Wilkinson transform.
p?org2l/p?ung2l	s,d,c,z	Generates all or part of the orthogonal/unitary matrix *Q* from a *QL* factorization determined by p?geqlf (unblocked algorithm).
p?org2r/p?ung2r	s,d,c,z	Generates all or part of the orthogonal/unitary matrix *Q* from a *QR* factorization determined by p?geqrf (unblocked algorithm).
p?orgl2/p?ungl2	s,d,c,z	Generates all or part of the orthogonal/unitary matrix *Q* from an *LQ* factorization determined by p?gelqf (unblocked algorithm).
p?orgr2/p?ungr2	s,d,c,z	Generates all or part of the orthogonal/unitary matrix *Q* from an *RQ* factorization determined by p?gerqf (unblocked algorithm).
p?orm2l/p?unm2l	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a *QL* factorization determined by p?geqlf (unblocked algorithm).
p?orm2r/p?unm2r	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a *QR* factorization determined by p?geqrf (unblocked algorithm).
p?orml2/p?unml2	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from an *LQ* factorization determined by p?gelqf (unblocked algorithm).
p?ormr2/p?unmr2	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from an *RQ* factorization determined by p?gerqf (unblocked algorithm).

Routine Name	Data Types	Description
p?pbtrsv	s,d,c,z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf.
p?pttrsv	s,d,c,z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf.
p?potf2	s,d,c,z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).
p?rot	s,d	Applies a planar rotation to two distributed vectors.
p?rscl	s,d,cs,zd	Multiplies a vector by the reciprocal of a real scalar.
p?sygs2/p?hegs2	s,d,c,z	Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).
p?sytd2/p?hetd2	s,d,c,z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).
p?trord	s,d	Reorders the Schur factorization of a general matrix.
p?trsen	s,d	Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.
p?trti2	s,d,c,z	Computes the inverse of a triangular matrix (local unblocked algorithm).
?lamsh	s,d	Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.
?laqr6	s,d	Performs a single small-bulge multi-shift QR sweep collecting the transformations.
?lar1va	s,d	*Computes scaled eigenvector corresponding to given eigenvalue.*
?laref	s,d	Applies Householder reflectors to matrices on either their rows or columns.
?larrb2	s,d	Provides limited bisection to locate eigenvalues for more accuracy.
?larrd2	s,d	Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.
?larre2	s,d	Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.
?larre2a	s,d	*Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.*
?larrf2	s,d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrv2	s,d	Computes the eigenvectors of the tridiagonal matrix $T = L*D*L^T$ given L, D and the eigenvalues of $L*D*L^T$.

Routine Name	Data Types	Description
?lasorte	s,d	Sorts eigenpairs by real and complex data types.
?lasrt2	s,d	Sorts numbers in increasing or decreasing order.
?stegr2	s,d	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
?stegr2a	s,d	Computes selected eigenvalues and initial representations needed for eigenvector computations.
?stegr2b	s,d	From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.
?stein2	s,d	Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.
?dbtf2	s,d,c,z	Computes an *LU* factorization of a general band matrix with no pivoting (local unblocked algorithm).
?dbtrf	s,d,c,z	Computes an *LU* factorization of a general band matrix with no pivoting (local blocked algorithm).
?dttrf	s,d,c,z	Computes an *LU* factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).
?dttrsv	s,d,c,z	Solves a general tridiagonal system of linear equations using the *LU* factorization computed by ?dttrf.
?pttrsv	s,d,c,z	Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the LDL^H factorization computed by ?pttrf.
?steqr2	s,d	Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit *QL* or *QR* method.
?trmvt	s,d,c,z	Performs matrix-vector operations.
pilaenv	NA	Returns the positive integer value of the logical blocking size.
pilaenvx	NA	Called from the ScaLAPACK routines to choose problem-dependent parameters for the local environment.
pjlaenv	NA	Called from the ScaLAPACK symmetric and Hermitian tailored eigen-routines to choose problem-dependent parameters for the local environment.

b?laapp

Multiplies a matrix with an orthogonal matrix.

Syntax

```
call bslaapp( iside, m, n, nb, a, lda, nitraf, itraf, dtraf, work )
```

```
call bdlaapp( iside, m, n, nb, a, lda, nitraf, itraf, dtraf, work )
```

Description

b?laapp computes

$B = Q^T A$ or $B = AQ$

where A is an m-by-n matrix and Q is an orthogonal matrix represented by the parameters in the arrays *itraf* and *dtraf* as described in b?trexc.

This is an auxiliary routine called by p?trord.

Input Parameters

iside — INTEGER

Specifies whether Q multiplies A from the left or right as follows:

= 0: compute $B = Q^T A$;

= 1: compute $B = AQ$.

m — INTEGER

The number of rows of A.

n — INTEGER

The number of columns of A.

nb — INTEGER

If *iside* = 0, the Q is applied block column-wise to the rows of A and *nb* specifies the maximal width of the block columns.

If *iside* = 1, this variable is not referenced.

a — REAL for bslaapp

DOUBLE PRECISION for bdlaapp

Array of size (*lda*,*n*).

On entry, the matrix A.

lda — INTEGER

The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

nitraf — INTEGER

Length of the array *itraf*. *nitraf* ≥ 0.

itraf — INTEGER array, length *nitraf*

List of parameters for representing the transformation matrix Q, see b?trexc.

work — (workspace) REAL array of size *n*.

OUTPUT Parameters

a — *a* is overwritten by B.

dtraf — REAL for bslaapp

DOUBLE PRECISION for bdlaapp

Array, length *k*.

If *iside*=0, *k* = 3*(*n*/*nb*).

If *iside*=1, *k* = 3**nitraf*.

List of parameters for representing the transformation matrix *Q*, see b?trexc

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

b?laexc

Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.

Syntax

```
call bslaexc( n, t, ldt, j1, n1, n2, itraf, dtraf, work, info )
call bdlaexc( n, t, ldt, j1, n1, n2, itraf, dtraf, work, info )
```

Description

b?laexc swaps adjacent diagonal blocks *T11* and *T22* of order 1 or 2 in an upper quasi-triangular matrix *T* by an orthogonal similarity transformation.

In contrast to the LAPACK routine ?laexc, the orthogonal transformation matrix *Q* is not explicitly constructed but represented by parameters contained in the arrays *itraf* and *dtraf*. See the description of b?trexc for more details.

T must be in Schur canonical form, that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

n

INTEGER

The order of the matrix *T*. *n*≥ 0.

t

REAL for bslaexc

DOUBLE PRECISION for bdlaexc

Array of size (*ldt*,*n*).

The upper quasi-triangular matrix *T*, in Schur canonical form.

ldt

INTEGER

The leading dimension of the array *t*. *ldt*≥ max(1,*n*).

j1

INTEGER

The index of the first row of the first block *T11*.

n1

INTEGER

The order of the first block *T11*. *n1* = 0, 1 or 2.

n2 INTEGER

The order of the second block *T22*. *n2* = 0, 1 or 2.

work REAL for bslaexc

DOUBLE PRECISION for bdlaexc

(Workspace) array of size *n*.

OUTPUT Parameters

t The updated matrix *T*, in Schur canonical form.

itraf INTEGER array, length *k*, where

k = 1, if *n1*+*n2* = 2;

k = 2, if *n1*+*n2* = 3;

k = 4, if *n1*+*n2* = 4.

List of parameters for representing the transformation matrix *Q*, see b?trexc.

dtraf REAL for bslaexc

DOUBLE PRECISION for bdlaexc

Array, length *k*, where

k = 2, if *n1*+*n2* = 2;

k = 5, if *n1*+*n2* = 3;

k = 10, if *n1*+*n2* = 4.

List of parameters for representing the transformation matrix *Q*, see b?trexc.

info INTEGER

= 0: successful exit

= 1: the transformed matrix *T* would be too far from Schur form; the blocks are not swapped and *T* and *Q* are unchanged.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

b?trexc

Reorders the Schur factorization of a general matrix.

Syntax

```
call bstrexc( n, t, ldt, ifst, ilst, nitraf, itraf, ndtraf, dtraf, work, info )
call bdtrexc( n, t, ldt, ifst, ilst, nitraf, itraf, ndtraf, dtraf, work, info )
```

Description

b?trexc reorders the real Schur factorization of a real matrix $A = Q * T * Q^T$, so that the diagonal block of *T* with row index *ifst* is moved to row *ilst*.

The real Schur form *T* is reordered by an orthogonal similarity transformation $Z^T * T * Z$. In contrast to the LAPACK routine ?trexc, the orthogonal matrix *Z* is not explicitly constructed but represented by parameters contained in the arrays *itraf* and *dtraf*. See Application Notes for further details.

T must be in Schur canonical form (as returned by ?hseqr), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

n

INTEGER

The order of the matrix *T*. $n \geq 0$.

t

REAL for bstrexc

DOUBLE PRECISION for bdtrexc

Array of size (*ldt*,*n*).

The upper quasi-triangular matrix *T*, in Schur canonical form.

ldt

INTEGER

The leading dimension of the array *t*. $ldt \geq \max(1,n)$.

ifst, *ilst*

INTEGER

Specify the reordering of the diagonal blocks of *T*. The block with row index *ifst* is moved to row *ilst*, by a sequence of transpositions between adjacent blocks.

nitraf

INTEGER

Length of the array *itraf*.

As a minimum requirement, $nitraf \geq \max(1,|ilst-ifst|)$.

If there are 2-by-2 blocks in *t* then *nitraf* must be larger; a safe choice is $nitraf \geq \max(1,2*|ilst-ifst|)$.

ndtraf

INTEGER

Length of the array *dtraf*.

As a minimum requirement, $ndtraf \geq \max(1,2*|ilst-ifst|)$.

If there are 2-by-2 blocks in *t* then *ndtraf* must be larger; a safe choice is $ndtraf \geq \max(1,5*|ilst-ifst|)$.

work

REAL for bstrexc

DOUBLE PRECISION for bdtrexc

(Workspace) array of size *n*.

OUTPUT Parameters

t

On exit, the reordered upper quasi-triangular matrix, in Schur canonical form.

ifst, *ilst*

If *ifst* pointed on entry to the second row of a 2-by-2 block, it is changed to point to the first row; *ilst* always points to the first row of the block in its final position (which may differ from its input value by +1 or -1).

	1 ≤*ifst*≤*n*; 1 ≤*ilst*≤*n*.
nitraf	Actual length of the array *itraf*.
itraf	INTEGER array, length *nitraf* List of parameters for representing the transformation matrix *Z*. See Application Notes for further details.
ndtraf	Actual length of the array *dtraf*.
dtraf	REAL for bstrexc DOUBLE PRECISION for bdtrexc Array, length *ndtraf* List of parameters for representing the transformation matrix *Z*. See Application Notes for further details.
info	INTEGER = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value = 1: two adjacent blocks were too close to swap (the problem is very ill-conditioned); *t* may have been partially reordered, and *ilst* points to the first row of the current position of the block being moved. = 2: the 2 by 2 block to be reordered split into two 1 by 1 blocks and the second block failed to swap with an adjacent block. *ilst* points to the first row of the current position of the whole block being moved.

Application Notes

The orthogonal transformation matrix *Z* is a product of *nitraf* elementary orthogonal transformations. The parameters defining these transformations are stored in the arrays *itraf* and *dtraf* as follows:

Consider the *i*-th transformation acting on rows/columns *pos*, *pos*+1, ... If this transformation is

- a Givens rotation with cosine *c* and sine *s* then

  itraf(*i*)= *pos*, *dtraf*(*i*) = *c*, *dtraf*(*i*+1) = *s*;
- a Householder reflector *H* = *I* - *t* * *v* * *v*' with *v* = [1; *v2*; *v3*] then

  itraf(*i*) = *n* + *pos*, *dtraf*(*i*) = *t*, *dtraf*(*i*+1) = *v2*, *dtraf*(*i*+2) = v3;
- a Householder reflector H = I - *t** *v* * *v*' with *v* = [*v1*; *v2*; 1] then

  itraf(*i*) = 2**n* + *pos*, *dtraf*(*i*) = *v1*, *dtraf*(*i*+1) = *v2*, *dtraf*(*i*+2) = *t*;

Note that the parameters in *dtraf* are stored consecutively.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lacgv

Conjugates a complex vector.

Syntax

```
call pclacgv(n, x, ix, jx, descx, incx)
call pzlacgv(n, x, ix, jx, descx, incx)
```

Description

The p?lacgvroutine conjugates a complex vector sub(*X*) of length *n*, where sub(*X*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*:*ix*+*n*-1, *jx*) if *incx* = 1.

Input Parameters

n — (global) INTEGER. The length of the distributed vector sub(*X*).

x — (local).

COMPLEX for pclacgv

COMPLEX*16 for pzlacgv.

Pointer into the local memory to an array of size (*lld_x*,*). On entry the vector to be conjugated *x*(i) = *X*(*ix*+(*jx*-1)*m_x+(*i*-1)**incx*), 1 ≤ *i* ≤ *n*.

ix — (global) INTEGER. The row index in the global matrix *X* indicating the first row of sub(*X*).

jx — (global) INTEGER. The column index in the global matrix *X* indicating the first column of sub(*X*).

descx — (global and local) INTEGER. Array of size *dlen_*=9. The array descriptor for the distributed matrix *X*.

incx — (global) INTEGER. The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

x — (local).

On exit, the local pieces of conjugated distributed vector sub(*X*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?max1

Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS p?amax, but using the absolute value to the real part).

Syntax

```
call pcmax1(n, amax, indx, x, ix, jx, descx, incx)
call pzmax1(n, amax, indx, x, ix, jx, descx, incx)
```

Description

The p?max1routine computes the global index of the maximum element in absolute value of a distributed vector sub(*X*). The global index is returned in *indx* and the value is returned in *amax*, where sub(*X*) denotes *X*(*ix*:*ix*+*n*-1, *jx*) if *incx* = 1, *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*.

Input Parameters

n	(global) pointer to INTEGER. The number of components of the distributed vector sub(*X*). $n \geq 0$.
x	(local) COMPLEX for pcmax1. COMPLEX*16 for pzmax1 Pointer into the local memory to an array of size (*lld_x*, *LOCc*(*jx*+*n*-1)). On entry this array contains the local pieces of the distributed vector sub(*X*).
ix	(global) INTEGER. The row index in the global matrix *X* indicating the first row of sub(*X*).
jx	(global) INTEGER. The column index in the global matrix *X* indicating the first column of sub(*X*).
descx	(global and local) INTEGER. Array of size *dlen_*. The array descriptor for the distributed matrix *X*.
incx	(global)INTEGER.The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

amax	(global output) pointer to REAL.The absolute value of the largest entry of the distributed vector sub(*X*) only in the scope of sub(*X*).
indx	(global output) pointer to INTEGER.The global index of the element of the distributed vector sub(*X*) whose real part has maximum absolute value.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

pilaver

Returns the ScaLAPACK version.

Syntax

```
call pilaver (vers_major, vers_minor, vers_patch )
```

Description

This subroutine returns the ScaLAPACK version.

Output Parameters

vers_major	INTEGER. Return the ScaLAPACK major version.
vers_minor	INTEGER. Return the ScaLAPACK minor version from the major version.
vers_patch	INTEGER.

Return the ScaLAPACK patch version from the minor version.

pmpcol

Finds the collaborators of a process.

Syntax

```
call pmpcol( myproc, nprocs, iil, needil, neediu, pmyils, pmyius, colbrt, frstcl,
lastcl )
```

Description

Using the output from pmpim2 and given the information on eigenvalue clusters, pmpcol finds the collaborators of *myproc*.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

Input Parameters

myproc INTEGER

The processor number, 0 ≤*myproc* < *nprocs*.

nprocs INTEGER

The total number of processors available.

iil INTEGER

The index of the leftmost eigenvalue in the eigenvalue cluster.

needil INTEGER

The leftmost position in the eigenvalue cluster needed by *myproc*.

neediu INTEGER

The rightmost position in the eigenvalue cluster needed by *myproc*.

pmyils INTEGER array

For each processor *p*, 0 < *p*≤*nprocs*, *pmyils*(*p*) is the index of the first eigenvalue in the eigenvalue cluster to be computed.

pmyils(*p*) equals zero if *p* stays idle.

pmyius INTEGER array

For each processor *p*, *pmyius*(*p*) is the index of the last eigenvalue in the eigenvalue cluster to be computed.

pmyius(*p*) equals zero if *p* stays idle.

OUTPUT Parameters

colbrt LOGICAL

Equals .TRUE. if *myproc* collaborates.

frstcl, *lastcl*

INTEGER

First and last collaborator of *myproc* .

myproc collaborates with:

frstcl, ..., *myproc*-1, *myproc*+1, ...,*lastcl*

If *myproc* = *frstcl*, there are no collaborators on the left. If *myproc* = *lastcl*, there are no collaborators on the right.

If *frstcl* = 0 and *lastcl* = *nprocs*-1, then *myproc* collaborates with everybody

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

pmpim2

Computes the eigenpair range assignments for all processes.

Syntax

```
call pmpim2( il, iu, nprocs, pmyils, pmyius )
```

Description

pmpim2 is the scheduling subroutine. It computes for all processors the eigenpair range assignments.

Input Parameters

il, *iu*

INTEGER

The range of eigenpairs to be computed.

nprocs

INTEGER

The total number of processors available.

Output Parameters

pmyils

INTEGER array

For each processor *p*, *pmyils*(*p*) is the index of the first eigenvalue in a cluster to be computed.

pmyils(*p*) equals zero if *p* stays idle.

pmyius

INTEGER array

For each processor *p*, *pmyius*(*p*) is the index of the last eigenvalue in a cluster to be computed.

pmyius(*p*) equals zero if *p* stays idle.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?combamax1

Finds the element with maximum real part absolute value and its corresponding global index.

Syntax

```
call ccombamax1(v1, v2)
call zcombamax1(v1, v2)
```

Description

The ?combamax1routine finds the element having maximum real part absolute value as well as its corresponding global index.

Input Parameters

v1 (local)

COMPLEX for ccombamax1

COMPLEX*16 for zcombamax1

Array of size 2. The first maximum absolute value element and its global index. *v1*(1)=amax, *v1*(2)=indx.

v2 (local)

COMPLEX for ccombamax1

COMPLEX*16 for zcombamax1

Array of size 2. The second maximum absolute value element and its global index. *v2*(1)=amax, *v2*(2)=indx.

Output Parameters

v1 (local).

The first maximum absolute value element and its global index. *v1*(1)=amax, *v1*(2)=indx.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?sum1

Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum*, but using the true absolute value.*

Syntax

```
call pscsum1(n, asum, x, ix, jx, descx, incx)
call pdzsum1(n, asum, x, ix, jx, descx, incx)
```

Description

The p?sum1routine returns the sum of absolute values of a complex distributed vector sub(*x*) in *asum*, where sub(*x*) denotes *X*(*ix*:*ix*+*n*-1, *jx*:*jx*), if *incx* = 1, *X*(*ix*:*ix*, *jx*:*jx*+*n*-1), if *incx* = *m_x*.

Based on p?asum from the Level 1 PBLAS. The change is to use the 'genuine' absolute value.

Input Parameters

n	(global) pointer to INTEGER. The number of components of the distributed vector sub(*x*). $n \geq 0$.
x	(local) COMPLEX for pscsum1 COMPLEX*16 for pdzsum1. Pointer into the local memory to an array of size (*lld_x*, *LOCc*(*jx*+*n*-1)). This array contains the local pieces of the distributed vector sub(*X*).
ix	(global) INTEGER.The row index in the global matrix *X* indicating the first row of sub(*X*).
jx	(global) INTEGER. The column index in the global matrix *X* indicating the first column of sub(*X*)
descx	(local) INTEGER. Array of size *dlen_*=9. The array descriptor for the distributed matrix *X*.
incx	(global) INTEGER.The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*.

Output Parameters

asum	(local) Pointer to REAL. The sum of absolute values of the distributed vector sub(*X*) only in its scope.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrsv

Computes an LU factorization of a general triangular matrix with no pivoting. The routine is called by p?dbtrs.

Syntax

```
call psdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pddbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Description

The p?dbtrsvroutine solves a banded triangular system of linear equations

A(1 :*n*, *ja*:*ja*+*n*-1) * *X* = *B*(*ib*:*ib*+*n*-1, 1 :*nrhs*) or

A(1 :*n*, *ja*:*ja*+*n*-1)^T * *X* = *B*(*ib*:*ib*+*n*-1, 1 :*nrhs*) (for real flavors); *A*(1 :*n*, *ja*:*ja*+*n*-1)^H* *X* = *B*(*ib*:*ib*+*n*-1, 1 :*nrhs*) (for complex flavors),

where *A*(1 :*n*, *ja*:*ja*+*n*-1) is a banded triangular matrix factor produced by the Gaussian elimination code of p?dbtrf and is stored in *A*(1 :*n*, *ja*:*ja*+*n*-1) and *af*. The matrix stored in *A*(1 :*n*, *ja*:*ja*+*n*-1) is either upper or lower triangular according to *uplo*, and the choice of solving *A*(1 :*n*, *ja*:*ja*+*n*-1) or *A*(1 :*n*, *ja*:*ja*+*n*-1)^T is dictated by the user by the parameter *trans*.

The routine p?dbtrf must be called first.

Input Parameters

uplo

(global) CHARACTER.

If *uplo*='U', the upper triangle of *A*(1:*n*, *ja*:*ja*+*n*-1) is stored,

if *uplo* = 'L', the lower triangle of *A*(1:*n*, *ja*:*ja*+*n*-1) is stored.

trans

(global) CHARACTER.

If *trans* = 'N', solve with *A*(1:*n*, *ja*:*ja*+*n*-1),

if *trans* = 'C', solve with conjugate transpose *A*(1:*n*, *ja*:*ja*+*n*-1).

n

(global) INTEGER. The order of the distributed submatrix *A*;(*n*≥ 0).

bwl

(global) INTEGER. Number of subdiagonals. 0 ≤ *bwl* ≤ *n*-1.

bwu

(global) INTEGER. Number of subdiagonals. 0 ≤ *bwu* ≤ *n*-1.

nrhs

(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix *B* (*nrhs*≥ 0).

a

(local).

REAL for psdbtrsv

DOUBLE PRECISION for pddbtrsv

COMPLEX for pcdbtrsv

COMPLEX*16 for pzdbtrsv.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)), where *lld_a*≥(*bwl*+*bwu*+1). On entry, this array contains the local pieces of the *n*-by-*n* unsymmetric banded distributed Cholesky factor *L* or L^T, represented in global *A* as *A*(1 :*n*, *ja*:*ja*+*n*-1). This local portion is stored in the packed banded format used in LAPACK. See the *Application Notes* below and the ScaLAPACK manual for more detail on the format of distributed matrices.

ja

(global) INTEGER. The index in the global matrix *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen_*.

if 1*d* type (*dtype_a* = 501 or 502), *dlen*≥ 7;

if 2*d* type (*dtype_a* = 1), *dlen*≥ 9. The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.

b

(local)

REAL for psdbtrsv

DOUBLE PRECISION for pddbtrsv

COMPLEX for pcdbtrsv

COMPLEX*16 for pzdbtrsv.

Pointer into the local memory to an array of local lead dimension *lld_b*≥*nb*. On entry, this array contains the local pieces of the right-hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

ib

(global) INTEGER. The row index in the global matrix *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb

(global and local) INTEGER array of size *dlen_*.

if 1*d* type (*dtype_b* =502), *dlen*≥7;

if 2*d* type (*dtype_b* =1), *dlen*≥9. The array descriptor for the distributed matrix *B*. Contains information of mapping *B* to memory.

laf

(local)

INTEGER. Size of user-input auxiliary fill-in space *af*.

laf≥*nb**(*bwl*+*bwu*)+6*max(*bwl*, *bwu*)*max(*bwl*, *bwu*). If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

work

(local).

REAL for psdbtrsv

DOUBLE PRECISION for pddbtrsv

COMPLEX for pcdbtrsv

COMPLEX*16 for pzdbtrsv.

Temporary workspace. This space may be overwritten in between calls to routines.

work must be the size given in *lwork*.

lwork

(local or global) INTEGER.

Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

lwork≥ max(*bwl*, *bwu*)**nrhs*.

Output Parameters

a

(local).

This local portion is stored in the packed banded format used in LAPACK. Please see the ScaLAPACK manual for more detail on the format of distributed matrices.

b — On exit, this contains the local piece of the solutions distributed matrix *X*.

af — (local).

`REAL` for `psdbtrsv`

`DOUBLE PRECISION` for `pddbtrsv`

`COMPLEX` for `pcdbtrsv`

`COMPLEX*16` for `pzdbtrsv`.

auxiliary fill-in space. The fill-in space is created in a call to the factorization routine `p?dbtrf` and is stored in *af*. If a linear system is to be solved using `p?dbtrf` after the factorization routine, *af* must not be altered after the factorization.

work — On exit, *work*`(1)` contains the minimal *lwork*.

info — (local).

`INTEGER`. If *info* = 0, the execution is successful.

< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info*`= - (`*i*`*100+`*j*`)`, if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dttrsv

Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by `p?dttrs`.

Syntax

```
call psdttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pddttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcdttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzdttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Description

The `p?dttrsv`routine solves a tridiagonal triangular system of linear equations

A`(1 :`*n*`, `*ja*`:`*ja*`+`*n*`-1)*`*X* `= `*B*`(`*ib*`:`*ib*`+`*n*`-1, 1 :`*nrhs*`)` or

A`(1 :`*n*`, `*ja*`:`*ja*`+`*n*`-1)`^{T} `* `*X* `= `*B*`(`*ib*`:`*ib*`+`*n*`-1, 1 :`*nrhs*`)` for real flavors; *A*`(1 :`*n*`, `*ja*`:`*ja*`+`*n*`-1)`^{H}`* `*X* `= `*B*`(`*ib*`:`*ib*`+`*n*`-1, 1 :`*nrhs*`)` for complex flavors,

where A(1 :n, ja:ja+n-1) is a tridiagonal matrix factor produced by the Gaussian elimination code of p?dttrf and is stored in A(1 :n, ja:ja+n-1) and *af*.

The matrix stored in A(1 :n, ja:ja+n-1) is either upper or lower triangular according to *uplo*, and the choice of solving A(1 :n, ja:ja+n-1) or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter *trans*.

The routine p?dttrf must be called first.

Input Parameters

uplo

(global) CHARACTER.

If *uplo*='U', the upper triangle of A(1:n, ja:ja+n-1) is stored,

if *uplo* = 'L', the lower triangle of *A*(1:*n*, ja:ja+n-1) is stored.

trans

(global) CHARACTER.

If *trans* = 'N', solve with A(1:n, ja:ja+n-1),

if *trans* = 'C', solve with conjugate transpose A(1:n, ja:ja+n-1).

n

(global) INTEGER. The order of the distributed submatrix *A*; (*n*≥ 0).

nrhs

(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B(*ib*:*ib*+*n*-1, 1:*nrhs*). (*nrhs*≥ 0).

dl

(local).

REAL for psdttrsv

DOUBLE PRECISION for pddttrsv

COMPLEX for pcdttrsv

COMPLEX*16 for pzdttrsv.

Pointer to local part of global vector storing the lower diagonal of the matrix.

Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*.

Must be of size ≥*nb_a*.

d

(local).

REAL for psdttrsv

DOUBLE PRECISION for pddttrsv

COMPLEX for pcdttrsv

COMPLEX*16 for pzdttrsv.

Pointer to local part of global vector storing the main diagonal of the matrix.

du

(local).

REAL for psdttrsv

DOUBLE PRECISION for pddttrsv

COMPLEX for pcdttrsv

COMPLEX*16 for pzdttrsv.

Pointer to local part of global vector storing the upper diagonal of the matrix.

Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja

(global) INTEGER. The index in the global matrix *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen_*.

if 1*d* type (*dtype_a* = 501 or 502), *dlen*≥ 7;

if 2*d* type (*dtype_a* = 1), *dlen*≥ 9.

The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.

b

(local)

REAL for psdttrsv

DOUBLE PRECISION for pddttrsv

COMPLEX for pcdttrsv

COMPLEX*16 for pzdttrsv.

Pointer into the local memory to an array of local lead dimension *lld_b*≥*nb*. On entry, this array contains the local pieces of the right-hand sides *B*(*ib*:*ib*+*n*-1, 1 :*nrhs*).

ib

(global) INTEGER. The row index in the global matrix *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb

(global and local) INTEGER array of size *dlen_*.

if 1*d* type (*dtype_b* = 502), *dlen*≥7;

if 2*d* type (*dtype_b* = 1), *dlen*≥ 9.

The array descriptor for the distributed matrix *B*. Contains information of mapping *B* to memory.

laf

(local).

INTEGER.

Size of user-input auxiliary fill-in space *af*.

laf≥ 2*(*nb*+2). If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

work

(local).

REAL for psdttrsv

DOUBLE PRECISION for pddttrsv

COMPLEX for pcdttrsv

COMPLEX*16 for pzdttrsv.

Temporary workspace. This space may be overwritten in between calls to routines.

work must be the size given in *lwork*.

lwork

(local or global) INTEGER.

Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

lwork≥ 10*npcol+4*nrhs.

Output Parameters

dl

(local).

On exit, this array contains information containing the factors of the matrix.

d

On exit, this array contains information containing the factors of the matrix. Must be of size ≥*nb_a*.

b

On exit, this contains the local piece of the solutions distributed matrix X.

af

(local).

REAL for psdttrsv

DOUBLE PRECISION for pddttrsv

COMPLEX for pcdttrsv

COMPLEX*16 for pzdttrsv.

Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?dttrf and is stored in *af*. If a linear system is to be solved using p?dttrs after the factorization routine, *af* must not be altered after the factorization.

work

On exit, work(1) contains the minimal *lwork*.

info

(local). INTEGER.

If *info*=0, the execution is successful.

if *info*< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gebal

Balances a general real/complex matrix.

Syntax

```
call psgebal( job, n, a, desca, ilo, ihi, scale, info )
call pdgebal( job, n, a, desca, ilo, ihi, scale, info )
call pcgebal( job, n, a, desca, ilo, ihi, scale, info )call pzgebal( job, n, a, desca, ilo, ihi, scale, info )
```

Description

p?gebal balances a general real/complex matrix *A*. This involves, first, permuting *A* by a similarity transformation to isolate eigenvalues in the first 1 to *ilo*-1 and last *ihi*+1 to *n* elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns *ilo* to *ihi* to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrix, and improve the accuracy of the computed eigenvalues and/or eigenvectors.

Input Parameters

job

(global) CHARACTER*1

Specifies the operations to be performed on *a*:

= 'N': none: simply set *ilo* = 1, *ihi* = *n*, *scale*(*i*) = 1.0 for *i* = 1,...,*n*;

= 'P': permute only;

= 'S': scale only;

= 'B': both permute and scale.

n

(global) INTEGER

The order of the matrix *A* ($n \geq 0$).

a

REAL for psgebal

DOUBLE PRECISION for pdgebal

COMPLEX for pcgebal

COMPLEX DOUBLE for pzgebal

(local) Pointer into the local memory to an array of size (*lld_a*, $LOC_c(n)$)

This array contains the local pieces of global input matrix *A*.

desca

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

OUTPUT Parameters

a

On exit, *a* is overwritten by the balanced matrix.

If *job* = 'N', *a* is not referenced.

See Notes for further details.

ilo, *ihi*

(global) INTEGER

ilo and *ihi* are set to integers such that on exit matrix elements *A*(*i*,*j*) are zero if *i* > *j* and *j* = 1,...,*ilo*-1 or *i* = *ihi*+1,...,*n*.

If *job* = 'N' or 'S', *ilo* = 1 and *ihi* = *n*.

scale

REAL for psgebal and pcgebal

DOUBLE PRECISION for pdgebal and pzgebal

(global) array of size *n*.

Details of the permutations and scaling factors applied to `a`. If *pj* is the index of the row and column interchanged with row and column *j* and *dj* is the scaling factor applied to row and column *j*, then

`scale`(*j*) = *pj* for *j* = 1,...,`ilo`-1, `ihi`+1,..., n

`scale`(*j*) = *dj* for *j* = `ilo`,...,`ihi`

The order in which the interchanges are made is `n` to `ihi`+1, then 1 to `ilo`-1.

`info` (global) `INTEGER`

= 0: successful exit.

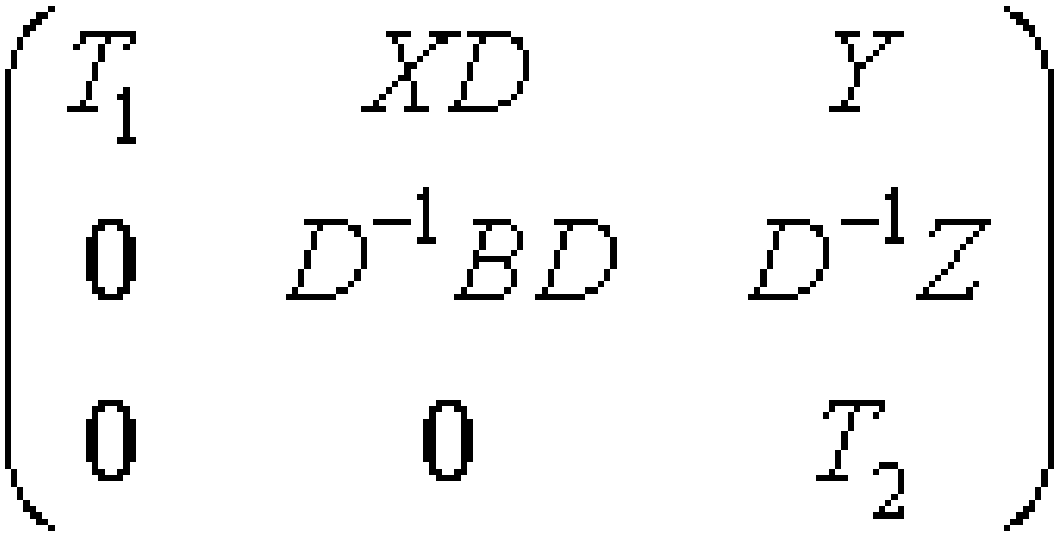
< 0: if `info` = -*i*, the *i*-th argument had an illegal value.

Application Notes

The permutations consist of row and column interchanges which put the matrix in the form

$$PAP = \begin{pmatrix} T_1 & X & Y \\ 0 & B & Z \\ 0 & 0 & T_2 \end{pmatrix}$$

where *T1* and *T2* are upper triangular matrices whose eigenvalues lie along the diagonal. The column indices `ilo` and `ihi` mark the starting and ending columns of the submatrix B. Balancing consists of applying a diagonal similarity transformation $D^{-1}BD$ to make the 1-norms of each row of *B* and its corresponding column nearly equal. The output matrix is



Information about the permutations *P* and the diagonal matrix *D* is returned in the vector `scale`.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gebd2

Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).

Syntax

```
call psgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pdgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pcgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pzgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

Description

The `p?gebd2`routine reduces a real/complex general *m*-by-*n* distributed matrix sub(*A*) = *A*`(ia:ia+m-1, ja:ja+n-1)` to upper or lower bidiagonal form *B* by an orthogonal/unitary transformation:

```
Q'*sub(A)*P = B.
```

If $m \geq n$, *B* is the upper bidiagonal; if $m<n$, *B* is the lower bidiagonal.

Input Parameters

`m`	(global) `INTEGER`. The number of rows of the distributed matrix sub(*A*). $(m \geq 0)$.
`n`	(global) `INTEGER`. The number of columns in the distributed matrix sub(*A*). ($n \geq 0$).
`a`	(local).

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Pointer into the local memory to an array of size (*lld_a*, LOC_c(*ja*+*n*-1)).

On entry, this array contains the local pieces of the general distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

work (local).

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

This is a workspace array of size *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork* ≥ max(*mpa*0, *nqa*0),

where *nb* = *mb_a* = *nb_a*, *iroffa* = mod(*ia*-1, *nb*),

iarow = indxg2p(*ia*, *nb*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb*, *mycol*, *csrc_a*, *npcol*),

*mpa*0 = numroc(*m*+*iroffa*, *nb*, *myrow*, *iarow*, *nprow*),

*nqa*0 = numroc(*n*+*icoffa*, *nb*, *mycol*, *iacol*, *npcol*).

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).

On exit, if *m* ≥ *n*, the diagonal and the first superdiagonal of sub(*A*) are overwritten with the upper bidiagonal matrix *B*; the elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the first superdiagonal, with the array *taup*, represent the orthogonal matrix *P* as a

product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix *B*; the elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the diagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors. See *Applications Notes* below.

d (local)

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array of size $LOCc(ja+\min(m,n)-1)$ if $m \geq n$; $LOCr(ia+\min(m,n)-1)$ otherwise. The distributed diagonal elements of the bidiagonal matrix *B*: $d(i) = a(i,i)$. *d* is tied to the distributed matrix *A*.

e (local)

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array of size $LOCc(ja+\min(m,n)-1)$ if $m \geq n$; $LOCr(ia+\min(m,n)-2)$ otherwise. The distributed diagonal elements of the bidiagonal matrix *B*:

if $m \geq n$, $e(i) = a(i, i+1)$ for $i = 1, 2, \ldots, n-1$;

if $m < n$, $e(i) = a(i+1, i)$ for $i = 1, 2, \ldots, m-1$. *e* is tied to the distributed matrix *A*.

tauq (local).

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array of size $LOCc(ja+\min(m,n)-1)$. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. *tauq* is tied to the distributed matrix *A*.

taup (local).

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array of size `LOCr(ia+min(m,n)-1)`. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *P*. *taup* is tied to the distributed matrix *A*.

work

On exit, `work(1)` returns the minimal and optimal *lwork*.

info

(local)

`INTEGER`.

If `info = 0`, the execution is successful.

if `info < 0`: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then `info` = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then `info` = -*i*.

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

If `m≥n`,

`Q = H(1)*H(2)*...*H(n), and P = G(1)*G(2)*...*G(n-1)`

Each *H*(i) and *G*(i) has the form:

`H(i) = I - tauq*v*v', and G(i) = I - taup*u*u',`

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors. `v(1: i-1) = 0`, `v(i) = 1`, and `v(i+i:m)` is stored on exit in

`A(ia+i-ia+m-1, ja+i-1);`

`u(1:i) = 0, u(i+1) = 1`, and `u(i+2:n)` is stored on exit in `A(ia+i-1, ja+i+1:ja+n-1);`

tauq is stored in `tauq(ja+i-1)` and *taup* in `taup(ia+i-1)`.

If `m < n`,

`v(1: i) = 0, v(i+1) = 1`, and `v(i+2:m)` is stored on exit in `A(ia+i+1: ia+m-1, ja+i-1);`

`u(1: i-1) = 0, u(i) = 1`, and `u(i+1 :n)` is stored on exit in `A(ia+i-1,ja+i:ja+n-1);`

tauq is stored in `tauq(ja+i-1)` and *taup* in `taup(ia+i-1)`.

The contents of sub(*A*) on exit are illustrated by the following examples:

$m = 6\ and\ n = 5 (m > n):$

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5\ and\ n = 6 (m < n):$

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *B*, *vi* denotes an element of the vector defining `H(i)`, and *ui* an element of the vector defining `G(i)`.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gehd2

Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).

Syntax

```
call psgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pdgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pcgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pzgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?gehd2routine reduces a real/complex general distributed matrix sub(*A*) to upper Hessenberg form *H* by an orthogonal/unitary similarity transformation: *Q*'*sub(*A*)**Q* = *H*, where sub(*A*) = *A*(*ia*+*n*-1 :*ia*+*n*-1, *ja*+*n*-1 :*ja*+*n*-1).

Input Parameters

n — (global) INTEGER. The order of the distributed submatrix *A*. (*n*≥ 0).

ilo, *ihi* — (global) INTEGER. It is assumed that the matrix sub(*A*) is already upper triangular in rows *ia*:*ia*+*ilo*-2 and *ia*+*ihi*:*ia*+*n*-1 and columns *ja*:*ja*+*jlo*-2 and *ja*+*jhi*:*ja*+*n*-1. See *Application Notes* for further information.

If *n*≥ 0, 1 ≤ *ilo* ≤ *ihi* ≤ *n*; otherwise set *ilo* = 1, *ihi* = *n*.

a — (local).

REAL for psgehd2

DOUBLE PRECISION for pdgehd2

COMPLEX for pcgehd2

COMPLEX*16 for pzgehd2.

Pointer into the local memory to an array of size (*lld_a*, LOC_c(*ja*+*n*-1)).

On entry, this array contains the local pieces of the *n*-by-*n* general distributed matrix sub(*A*) to be reduced.

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of sub(*A*), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work — (local).

REAL for psgehd2

DOUBLE PRECISION for pdgehd2

COMPLEX for pcgehd2

COMPLEX*16 for pzgehd2.

This is a workspace array of size *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork*≥*nb* + max(*npa*0, *nb*), where *nb* = *mb_a* = *nb_a*, *iroffa* = mod(*ia*-1, *nb*), *iarow* = indxg2p (*ia*, *nb*, *myrow*, *rsrc_a*, *nprow*),*npa*0 = numroc(*ihi* +*iroffa*, *nb*, *myrow*, *iarow*, *nprow*).

indxg2p and numroc are ScaLAPACK tool functions;*myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local). On exit, the upper triangle and the first subdiagonal of sub(*A*) are overwritten with the upper Hessenberg matrix *H*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. (see *Application Notes* below).

tau (local).

REAL for psgehd2

DOUBLE PRECISION for pdgehd2

COMPLEX for pcgehd2

COMPLEX*16 for pzgehd2.

Array of size *LOCc*(*ja*+*n*-2) The scalar factors of the elementary reflectors (see *Application Notes* below). Elements *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-2 of the global vector *tau* are set to zero. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.

If *info* = 0, the execution is successful.

if *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of (*ihi-ilo*) elementary reflectors

```
Q = H(ilo)*H(ilo+1)*...*H(ihi-1).
```

Each *H*(*i*) has the form

```
H(i) = I - tau*v*v',
```

where *tau* is a real/complex scalar, and *v* is a real/complex vector with `v(1: i)=0`, `v(i+1)=1` and `v(ihi +1:n)=0`; `v(i+2:ihi)` is stored on exit in `A(ia+ilo+i:ia+ihi-1, ia+ilo+i-2)`, and *tau* in `tau(ja+ilo +i-2)`.

The contents of `A(ia:ia+n-1, ja:ja+n-1)` are illustrated by the following example, with `n = 7`, `ilo = 2` and `ihi = 6`:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & & & & & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & h & h & h & h & a \\ & a & h & h & h & h & a \\ & h & h & h & h & h & h \\ & v2 & h & h & h & h & h \\ & v2 & v3 & h & h & h & h \\ & v2 & v3 & v4 & h & h & h \\ & & & & & & a \end{bmatrix}$$

where *a* denotes an element of the original matrix sub(*A*), *h* denotes a modified element of the upper Hessenberg matrix *H*, and *vi* denotes an element of the vector defining `H(ja+ilo+i-2)`.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gelq2

Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The `p?gelq2`routine computes an *LQ* factorization of a real/complex distributed *m*-by-*n* matrix `sub(A) = A(ia:ia+m-1, ja:ja+n-1) = L*Q`.

Input Parameters

`m` (global) `INTEGER`.

The number of rows of the distributed matrix sub(*A*). `(m≥0)`.

`n` (global) `INTEGER`.

The number of columns of the distributed matrix sub(*A*). `(n≥0)`.

`a` (local).

REAL for psgelq2

DOUBLE PRECISION for pdgelq2

COMPLEX for pcgelq2

COMPLEX*16 for pzgelq2.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work

(local).

REAL for psgelq2

DOUBLE PRECISION for pdgelq2

COMPLEX for pcgelq2

COMPLEX*16 for pzgelq2.

This is a workspace array of size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork*≥*nq*0 + max(1, *mp*0),

where *iroff* = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

(local).

On exit, the elements on and below the diagonal of sub(*A*) contain the *m* by min(*m*,*n*) lower trapezoidal matrix *L* (*L* is lower triangular if *m* ≤ *n*); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau	(local).
	`REAL` for `psgelq2`
	`DOUBLE PRECISION` for `pdgelq2`
	`COMPLEX` for `pcgelq2`
	`COMPLEX*16` for `pzgelq2`.
	Array of size *LOCr*(*ia*+min(*m*, *n*)-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work	On exit, *work*(1) returns the minimal and optimal *lwork*.
info	(local) `INTEGER`. If *info* = 0, the execution is successful. if *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q =*H*(*ia*+*k*-1)**H*(*ia*+*k*-2)*. . . **H*(*ia*) for real flavors, *Q* =(*H*(*ia*+*k*-1))^H*(*H*(*ia*+*k*-2))^H...*(*H*(*ia*))^H for complex flavors,

where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau***v***v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1: *i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+1: *n*) (for real flavors) or conjg(*v*(*i*+1: *n*)) (for complex flavors) is stored on exit in *A*(*ia*+*i*-1,*ja*+*i*:*ja*+*n*-1), and *tau* in *tau*(*ia*+*i*-1).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?geql2

Computes a QL factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?geql2routine computes a *QL* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1)= *Q* **L*.

Input Parameters

m	(global) `INTEGER`.

The number of rows in the distributed matrix sub(*A*). (*m*≥ 0).

n

(global) INTEGER.

The number of columns in the distributed matrix sub(*A*). (*n*≥ 0).

a

(local).

REAL for psgeql2

DOUBLE PRECISION for pdgeql2

COMPLEX for pcgeql2

COMPLEX*16 for pzgeql2.

Pointer into the local memory to an array of size (*lld_a*, $LOC_c(ja+n-1)$).

On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

work

(local).

REAL for psgeql2

DOUBLE PRECISION for pdgeql2

COMPLEX for pcgeql2

COMPLEX*16 for pzgeql2.

This is a workspace array of size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least lwork≥*mp*0 + max(1, *nq*0),

where *iroff* = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

(local).

On exit,

if $m \geq n$, the lower triangle of the distributed submatrix $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the n-by-n lower triangular matrix L;

if $m \leq n$, the elements on and below the $(n-m)$-th superdiagonal contain the m-by-n lower trapezoidal matrix L; the remaining elements, with the array *tau*, represent the orthogonal/ unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

tau (local).

`REAL` for `psgeql2`

`DOUBLE PRECISION` for `pdgeql2`

`COMPLEX` for `pcgeql2`

`COMPLEX*16` for `pzgeql2`.

Array of size *LOCc*(*ja*+*n*-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local). `INTEGER`.

If *info* = 0, the execution is successful. if *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix Q is represented as a product of elementary reflectors

Q = *H*(*ja*+*k*-1)*...**H*(*ja*+1)**H*(*ja*), where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H(*i*) = *I*- *tau* **v***v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*m*-*k*+*i*+1: *m*) = 0 and *v*(*m*-*k*+*i*) = 1; *v*(1: *m*-*k*+*i*-1) is stored on exit in *A*(*ia*:*ia*+*m*-*k*+*i*-2, *ja*+*n*-*k*+*i*-1), and *tau* in *tau*(*ja*+*n*-*k*+*i*-1).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?geqr2

Computes a QR factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?geqr2routine computes a *QR* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1)= *Q***R*.

Input Parameters

m — (global) INTEGER.

The number of rows in the distributed matrix sub(*A*). (*m*≥0).

n — (global) INTEGER. The number of columns in the distributed matrix sub(*A*). (*n*≥0).

a — (local).

REAL for psgeqr2

DOUBLE PRECISION for pdgeqr2

COMPLEX for pcgeqr2

COMPLEX*16 for pzgeqr2.

Pointer into the local memory to an array of size (*lld_a*, LOC_c(*ja*+*n*-1)).

On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of sub(*A*), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

work — (local).

REAL for psgeqr2

DOUBLE PRECISION for pdgeqr2

COMPLEX for pcgeqr2

COMPLEX*16 for pzgeqr2.

This is a workspace array of size *lwork*.

lwork — (local or global) INTEGER.

The size of the array *work*.

l*work* is local input and must be at least *lwork*≥*mp*0+max(1, *nq*0),

where *iroff* = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*).

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).

On exit, the elements on and above the diagonal of sub(*A*) contain the min(*m*,*n*) by *n* upper trapezoidal matrix *R* (*R* is upper triangular if *m*≥*n*); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau (local).

REAL for psgeqr2

DOUBLE PRECISION for pdgeqr2

COMPLEX for pcgeqr2

COMPLEX*16 for pzgeqr2.

Array of size *LOCc*(*ja*+min(*m*,*n*)-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work On exit, work(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.

If info = 0, the execution is successful. if *info* < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then info = - (*i**100+*j*),

if the *i*-th argument is a scalar and had an illegal value, then info = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ja*)**H*(*ja*+1)*. . .* *H*(*ja*+*k*-1), where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H(*j*)= *I* - *tau***v***v*',

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1: *i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+1: *m*) is stored on exit in *A*(*ia*+*i*:*ia*+*m*-1, *ja*+*i*-1), and *tau* in *tau*(*ja*+*i*-1).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gerq2

Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

call psgerq2(*m*, *n*, *a*, *ia*, *ja*, *desca*, *tau*, *work*, *lwork*, *info*)

```
call pdgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?gerq2routine computes an *RQ* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) = *R***Q*.

Input Parameters

m — (global) INTEGER. The number of rows in the distributed matrix sub(*A*). (*m*≥0).

n — (global) INTEGER. The number of columns in the distributed matrix sub(*A*). (*n*≥0).

a — (local).

REAL for psgerq2

DOUBLE PRECISION for pdgerq2

COMPLEX for pcgerq2

COMPLEX*16 for pzgerq2.

Pointer into the local memory to an array of size (*lld_a*, LOC_c(*ja*+*n*-1)).

On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of sub(*A*), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

work — (local).

REAL for psgerq2

DOUBLE PRECISION for pdgerq2

COMPLEX for pcgerq2

COMPLEX*16 for pzgerq2.

This is a workspace array of size *lwork*.

lwork — (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork*≥*nq*0 + max(1, *mp*0), where

iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*), *mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).

On exit,

if *m* ≤ *n*, the upper triangle of *A*(*ia*+*m*-*n*:*ia*+*m*-*1*, *ja*:*ja*+*n*-*1*) contains the *m*-by-*m* upper triangular matrix *R*;

if *m* ≥ *n*, the elements on and above the (*m*-*n*)-th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*; the remaining elements, with the array *tau*, represent the orthogonal/ unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau (local).

REAL for psgerq2

DOUBLE PRECISION for pdgerq2

COMPLEX for pcgerq2

COMPLEX*16 for pzgerq2.

Array of size *LOCr*(*ia*+*m* -1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.

If *info* = 0, the execution is successful.

if *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ia*)**H*(*ia*+1)*...**H*(*ia*+*k*-1) for real flavors,

$Q = (H(ia))^{H}*(H(ia+1))^{H}...*(H(ia+k-1))^{H}$ for complex flavors,

where *k* = min(*m*, *n*).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau***v***v*',

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*n*-*k*+*i*+1:*n*) = 0 and *v*(*n*-*k*+*i*) = 1; *v*(1:*n*-*k*+*i*-1) for real flavors or conjg(*v*(1:*n*-*k*+*i*-1)) for complex flavors is stored on exit in *A*(*ia*+*m*-*k*+*i*-1, *ja*:*ja*+*n*-*k*+*i*-2), and *tau* in *tau*(*ia*+*m*-*k*+*i*-1).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?getf2

Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).

Syntax

```
call psgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pdgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pcgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pzgetf2(m, n, a, ia, ja, desca, ipiv, info)
```

Description

The p?getf2routine computes an *LU* factorization of a general *m*-by-*n* distributed matrix sub(*A*) = `A(ia:ia+m-1, ja:ja+n-1)` using partial pivoting with row interchanges.

The factorization has the form `sub(A) = P * L* U`, where *P* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if $m>n$), and *U* is upper triangular (upper trapezoidal if $m < n$). This is the right-looking Parallel Level 2 BLAS version of the algorithm.

Input Parameters

m

(global) INTEGER.

The number of rows in the distributed matrix sub(*A*). $(m\geq 0)$.

n

(global) INTEGER. The number of columns in the distributed matrix sub(*A*). $(nb\_a - \mathrm{mod}(ja-1, nb\_a)\geq n\geq 0)$.

a

(local).

REAL for psgetf2

DOUBLE PRECISION for pdgetf2

COMPLEX for pcgetf2

COMPLEX*16 for pzgetf2.

Pointer into the local memory to an array of size $(lld\_a, LOC_c(ja+n-1))$.

On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

ipiv (local) INTEGER.

Array of size (*LOCr(m_a) + mb_a*). This array contains the pivoting information. *ipiv(i)* -> The global row that local row *i* was swapped with. This array is tied to the distributed matrix *A*.

info (local). INTEGER.

If *info* = 0: successful exit.

If *info* < 0:

- if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*),
- if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* > 0: If *info* = *k*, the matrix element *U*(*ia*+*k*-1, *ja*+*k*-1) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?labrd

Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.

Syntax

```
call pslabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
call pdlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
call pclabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
call pzlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
```

Description

The p?labrdroutine reduces the first *nb* rows and columns of a real/complex general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) to upper or lower bidiagonal form by an orthogonal/unitary transformation *Q'* A * P*, and returns the matrices *X* and *Y* necessary to apply the transformation to the unreduced part of sub(*A*).

If *m* ≥*n*, sub(*A*) is reduced to upper bidiagonal form; if *m* < *n*, sub(*A*) is reduced to lower bidiagonal form.

This is an auxiliary routine called by p?gebrd.

Input Parameters

m

(global) INTEGER. The number of rows in the distributed matrix sub(*A*). (*m*≥ 0).

n

(global) INTEGER. The number of columns in the distributed matrix sub(*A*). (*n* ≥ 0).

nb

(global) INTEGER.

The number of leading rows and columns of sub(*A*) to be reduced.

a

(local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the general distributed matrix sub(*A*).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

ix, *jx*

(global) INTEGER. The row and column indices in the global matrix *X* indicating the first row and the first column of the matrix sub(*X*), respectively.

descx

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *X*.

iy, *jy*

(global) INTEGER. The row and column indices in the global matrix *Y* indicating the first row and the first column of the matrix sub(*Y*), respectively.

descy

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Y*.

work

(local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Workspace array of size*lwork*.

lwork ≥ *nb_a* + *nq*,

with *nq* = numroc(*n*+mod(*ia*-1, *nb_y*), *nb_y*, *mycol*, *iacol*, *npcol*)

iacol = indxg2p (*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*)

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

a (local)

On exit, the first *nb* rows and columns of the matrix are overwritten; the rest of the distributed matrix sub(*A*) is unchanged.

If $m \geq n$, elements on and below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors; and elements above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors.

If $m < n$, elements below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements on and above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors. See *Application Notes* below.

d (local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Array of size *LOCr*(*ia*+min(*m*,*n*)-1) if $m \geq n$; *LOCc*(*ja*+min(*m*,*n*)-1) otherwise. The distributed diagonal elements of the bidiagonal distributed matrix *B*:

d(*i*) = *A*(*ia*+*i*-1, *ja*+*i*-1).

d is tied to the distributed matrix *A*.

e (local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Array of size *LOCr*(*ia*+min(*m*,*n*)-1) if $m \geq n$; *LOCc*(*ja*+min(*m*,*n*)-2) otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix *B*:

if $m \geq n$, *E*(*i*) = *A*(*ia*+*i*-1, *ja*+*i*) for *i* = 1, 2, ..., *n*-1;

if $m<n$, *E*(*i*) = *A*(*ia*+*i*, *ja*+*i*-1) for *i* = 1, 2, ..., *m*-1.

e is tied to the distributed matrix *A*.

tauq, *taup* (local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Array size *LOCc*(*ja*+min(*m*, *n*)-1) for *tauq*, size *LOCr*(*ia*+min(*m*, *n*)-1) for *taup*. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q* for *tauq*, *P* for *taup*. *tauq* and *taup* are tied to the distributed matrix *A*. See *Application Notes* below.

x (local)

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Pointer into the local memory to an array of size *lld_x* by *nb*. On exit, the local pieces of the distributed *m*-by-*nb* matrix *X*(*ix*:*ix*+*m*-1, *jx*:*jx*+*nb*-1) required to update the unreduced part of sub(*A*).

y (local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Pointer into the local memory to an array of size *lld_y* by *nb*. On exit, the local pieces of the distributed *n*-by-*nb* matrix *Y*(*iy*:*iy*+*n*-1, *jy*:*jy*+*nb*-1) required to update the unreduced part of sub(*A*).

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

Q = *H*(1)**H*(2)*...**H*(*nb*), and *P* = *G*(1)**G*(2)*...**G*(*nb*)

Each *H*(i) and *G*(i) has the form:

H(i) = *I* - *tauq***v***v*', and *G*(i) = *I* - *taup***u***u*',

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors.

If *m* ≥ *n*, *v*(1: *i*-1) = 0, *v*(*i*) = 1, and *v*(*i*:*m*) is stored on exit in

A(*ia*+*i*-1:*ia*+*m*-1, *ja*+*i*-1); *u*(1:*i*) = 0, *u*(*i*+1) = 1, and *u*(*i*+1:*n*) is stored on exit in *A*(*ia*+*i*-1, *ja*+*i*:*ja*+*n*-1); *tauq* is stored in *tauq*(*ja*+*i*-1) and *taup* in *taup*(*ia*+*i*-1).

If *m* < *n*, *v*(1: *i*) = 0, *v*(*i*+1) = 1, and *v*(*i*+1:*m*) is stored on exit in

A(*ia*+*i*+1:*ia*+*m*-1, *ja*+*i*-1); *u*(1:*i*-1) = 0, *u*(*i*) = 1, and *u*(*i*:*n*) is stored on exit in *A*(*ia*+*i*-1, *ja*+*i*:*ja*+*n*-1); *tauq* is stored in *tauq*(*ja*+*i*-1) and *taup* in *taup*(*ia*+*i*-1). The elements of the vectors *v* and *u* together form the *m*-by-*nb* matrix *V* and the *nb*-by-*n* matrix *U*' which are necessary, with *X* and *Y*, to apply the transformation to the unreduced part of the matrix, using a block update of the form: sub(*A*) := sub(*A*) - *V***Y*' - *X***U*'. The contents of sub(*A*) on exit are illustrated by the following examples with *nb* = 2:

$m = 6$ **and** $n = 5 (m > n)$:

$$\begin{bmatrix} 1 & 1 & u1 & u1 & u1 \\ v1 & 1 & 1 & u2 & u2 \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

$m = 5$ **and** $n = 6 (m < n)$ **:**

$$\begin{bmatrix} 1 & u1 & u1 & u1 & u1 & u1 \\ 1 & 1 & u2 & u2 & u2 & u2 \\ v1 & 1 & a & a & a & a \\ v1 & v2 & a & a & a & a \\ v1 & v2 & a & a & a & a \end{bmatrix}$$

where *a* denotes an element of the original matrix which is unchanged, *vi* denotes an element of the vector defining `H(i)`, and *ui* an element of the vector defining `G(i)`.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lacon

Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.

Syntax

```
call pslacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pdlacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pclacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pzlacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
```

Description

The `p?lacon`routine estimates the 1-norm of a square, real/unitary distributed matrix *A*. Reverse communication is used for evaluating matrix-vector products. *x* and *v* are aligned with the distributed matrix *A*, this information is implicitly contained within *iv*, *ix*, *descv*, and *descx*.

Input Parameters

n

(global) `INTEGER`. The length of the distributed vectors *v* and *x*. $n \geq 0$.

v

(local).

`REAL` for `pslacon`

`DOUBLE PRECISION` for `pdlacon`

`COMPLEX` for `pclacon`

`COMPLEX*16` for `pzlacon`.

Pointer into the local memory to an array of size *LOCr*(*n*+mod(*iv*-1, *mb_v*)). On the final return, `v = a*w`, where *est* = norm(*v*)/norm(*w*) (*w* is not returned).

iv, *jv*	(global) INTEGER. The row and column indices in the global matrix *V* indicating the first row and the first column of the submatrix *V*, respectively.
descv	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix V.
x	(local). REAL for pslacon DOUBLE PRECISION for pdlacon COMPLEX for pclacon COMPLEX*16 for pzlacon. Pointer into the local memory to an array of size *LOCr*(*n*+mod(*ix*-1, *mb_x*)).
ix, *jx*	(global) INTEGER. The row and column indices in the global matrix *X* indicating the first row and the first column of the submatrix *X*, respectively.
descx	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix X.
isgn	(local). INTEGER. Array of size *LOCr*(*n*+mod(*ix*-1, *mb_x*)). *isgn* is aligned with *x* and *v*.
kase	(local). INTEGER. On the initial call to p?lacon, *kase* should be 0.

Output Parameters

x	(local). On an intermediate return, X should be overwritten by *A***X*, if *kase*=1, *A*'**X*, if *kase*=2, p?lacon must be re-called with all the other parameters unchanged.
est	(global). REAL for single precision flavors DOUBLE PRECISION for double precision flavors
kase	(local) INTEGER. On an intermediate return, *kase* is 1 or 2, indicating whether *X* should be overwritten by *A*X,* or *A'*X*. On the final return from p?lacon, *kase* is again 0.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laconsb

Looks for two consecutive small subdiagonal elements.

Syntax

```
call pslaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
call pdlaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
call pclaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
```

```
call pzlaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
```

Description

The p?laconsbroutine looks for two consecutive small subdiagonal elements by analyzing the effect of starting a double shift *QR* iteration given by *h44*, *h33*, and *h43h34* to see if this process makes a subdiagonal negligible.

Input Parameters

a

(local) REAL for pslaconsb

DOUBLE PRECISION for pdlaconsb

COMPLEX for pclaconsb

DOUBLE COMPLEX for pzlaconsb

Array of size (*lld_a*,*LOCc*(*n_a*)). On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.

desca

(global and local) INTEGER.

Array of size *dlen_*. The array descriptor for the distributed matrix *A*.

i

(global) INTEGER.

The global location of the bottom of the unreduced submatrix of *A*. Unchanged on exit.

l

(global) INTEGER.

The global location of the top of the unreduced submatrix of *A*. Unchanged on exit.

h44, *h33*, *h43h34*

(global). REAL for pslaconsb

DOUBLE PRECISION for pdlaconsb

COMPLEX for pclaconsb

DOUBLE COMPLEX for pzlaconsb

These three values are for the double shift *QR* iteration.

lwork

(local) INTEGER.

This must be at least 7*ceil(ceil((*i*-l)/*mb_a*)/lcm(*nprow*, *npcol*)). Here lcm is the least common multiple and *nprow***npcol* is the logical grid size.

Output Parameters

m

(global). On exit, this yields the starting location of the *QR* double shift. This will satisfy:

$l \le m \le i-2$.

buf

(local).

REAL for pslaconsb

DOUBLE PRECISION for pdlaconsb

COMPLEX for pclaconsb

DOUBLE COMPLEX for pzlaconsb

Array of size *lwork*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lacp2

Copies all or part of a distributed matrix to another distributed matrix.

Syntax

```
call pslacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

The p?lacp2routine copies all or part of a distributed matrix *A* to another distributed matrix *B*. No communication is performed, p?lacp2 performs a local copy sub(*A*):= sub(*B*), where sub(*A*) denotes *A*(*ia*:*ia*+*m*-1, *a*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*m*-1, *jb*:*jb*+*n*-1).

p?lacp2 requires that only dimension of the matrix operands is distributed.

Input Parameters

uplo

(global) CHARACTER. Specifies the part of the distributed matrix sub(*A*) to be copied:

= 'U': Upper triangular part is copied; the strictly lower triangular part of sub(*A*) is not referenced;

= 'L': Lower triangular part is copied; the strictly upper triangular part of sub(*A*) is not referenced.

Otherwise: all of the matrix sub(*A*) is copied.

m

(global) INTEGER.

The number of rows in the distributed matrix sub(*A*). (*m*≥ 0).

n

(global) INTEGER.

The number of columns in the distributed matrix sub(*A*). (*n*≥ 0).

a

(local).

REAL for pslacp2

DOUBLE PRECISION for pdlacp2

COMPLEX for pclacp2

COMPLEX*16 for pzlacp2.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*).

ia, *ja*	(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of sub(*A*), respectively.
desca	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.
ib, *jb*	(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of sub(*B*), respectively.
descb	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Output Parameters

b	(local). REAL for pslacp2 DOUBLE PRECISION for pdlacp2 COMPLEX for pclacp2 COMPLEX*16 for pzlacp2. Pointer into the local memory to an array of size (*lld_b*, *LOCc*(*jb*+*n*-1)). This array contains on exit the local pieces of the distributed matrix sub(*B*) set as follows: if *uplo* = 'U', *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), 1≤*i*≤*j*, 1≤*j*≤*n*; if *uplo* = 'L', *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), *j*≤*i*≤*m*, 1≤*j*≤*n*; otherwise, *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), 1≤*i*≤*m*, 1≤*j*≤*n*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lacp3

Copies from a global parallel array into a local replicated array or vice versa.

Syntax

```
call pslacp3(m, i, a, desca, b, ldb, ii, jj, rev)
call pdlacp3(m, i, a, desca, b, ldb, ii, jj, rev)
call pclacp3(m, i, a, desca, b, ldb, ii, jj, rev)
call pzlacp3(m, i, a, desca, b, ldb, ii, jj, rev)
```

Description

This is an auxiliary routine that copies from a global parallel array into a local replicated array or vise versa. Note that the entire submatrix that is copied gets placed on one node or more. The receiving node can be specified precisely, or all nodes can receive, or just one row or column of nodes.

Product and Performance Information
Notice revision #20201201

Input Parameters

m

(global) INTEGER.

m is the order of the square submatrix that is copied.

m≥ 0. Unchanged on exit.

i

(global) INTEGER. *A*(*i*, *i*) is the global location that the copying starts from. Unchanged on exit.

a

(local) REAL for pslacp3

DOUBLE PRECISION for pdlacp3

COMPLEX for pclacp3

DOUBLE COMPLEX for pzlacp3

Array of size (*lld_a*,*LOCc*(*n_a*)). On entry, the parallel matrix to be copied into or from.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

b

(local)

REAL for pslacp3

DOUBLE PRECISION for pdlacp3

COMPLEX for pclacp3

DOUBLE COMPLEX for pzlacp3

Array of size (*lld_b*,*LOCc*(*m*)). If *rev* = 0, this is the global portion of the matrix *A*(*i*:*i*+*m*-1, *i*:*i*+*m*-1). If *rev* = 1, this is unchanged on exit.

ldb

(local)

INTEGER.

The leading dimension of *B*.

ii, *jj*

(global) INTEGER. By using *rev* 0 and 1, data can be sent out and returned again. If *rev* = 0, then *ii* is destination row index and *jj* is destination column index for the node(s) receiving the replicated *B*. If *ii* ≥ 0, *jj* ≥ 0, then node (*ii*, *jj*) receives the data. If *ii* = -1, *jj* ≥ 0, then all rows in column *jj* receive the data. If *ii* ≥ 0, *jj* = -1, then all cols in row *ii* receive the data. If *ii* = -1, *jj* = -1, then all nodes receive the data. If *rev* !=0, then *ii* is the source row index for the node(s) sending the replicated *B*.

rev

(global) INTEGER. Use *rev* = 0 to send global *A* into locally replicated *B* (on node (*ii*, *jj*)). Use *rev* != 0 to send locally replicated *B* from node (*ii*, *jj*) to its owner (which changes depending on its location in *A*) into the global *A*.

Output Parameters

a	On exit, if *rev* = 1, the copied data. Unchanged on exit if *rev* = 0.
b	If *rev* = 1, this is unchanged on exit.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call pslacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

The p?lacpyroutine copies all or part of a distributed matrix *A* to another distributed matrix *B*. No communication is performed, p?lacpy performs a local copy sub(*B*):= sub(*A*), where sub(*A*) denotes *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*m*-1,*jb*:*jb*+*n*-1).

Input Parameters

uplo	(global) CHARACTER. Specifies the part of the distributed matrix sub(*A*) to be copied: = 'U': Upper triangular part; the strictly lower triangular part of sub(*A*) is not referenced; = 'L': Lower triangular part; the strictly upper triangular part of sub(*A*) is not referenced. Otherwise: all of the matrix sub(*A*) is copied.
m	(global) INTEGER. The number of rows in the distributed matrix sub(*A*). (*m*≥0).
n	(global) INTEGER. The number of columns in the distributed matrix sub(*A*). (*n*≥0).
a	(local). REAL for pslacpy DOUBLE PRECISION for pdlacpy COMPLEX for pclacpy COMPLEX*16 for pzlacpy. Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

	On entry, this array contains the local pieces of the distributed matrix sub(*A*).
ia, *ja*	(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.
desca	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.
ib, *jb*	(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of sub(*B*) respectively.
descb	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

Output Parameters

b	(local). REAL for pslacpy DOUBLE PRECISION for pdlacpy COMPLEX for pclacpy COMPLEX*16 for pzlacpy. Pointer into the local memory to an array of size (*lld_b*, *LOCc*(*jb*+*n*-1)). This array contains on exit the local pieces of the distributed matrix sub(B) set as follows: if *uplo* = 'U', *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), 1≤*i*≤*j*, 1≤*j*≤*n*; if *uplo* = 'L', *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), *j*≤*i*≤*m*, 1≤*j*≤*n*; otherwise, *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), 1≤*i*≤*m*, 1≤*j*≤*n*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laevswp

Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.

Syntax

```
call pslaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pdlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pclaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pzlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
```

Description

The p?laevswproutine moves the eigenvectors (potentially unsorted) from where they are computed, to a ScaLAPACK standard block cyclic array, sorted so that the corresponding eigenvalues are sorted.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

n (global) INTEGER.

The order of the matrix *A*. $n \geq 0$.

zin (local).

REAL for pslaevswp

DOUBLE PRECISION for pdlaevswp

COMPLEX for pclaevswp

COMPLEX*16 for pzlaevswp.

Array of size (*ldzi*, *nvs*(*iam*+2)). The eigenvectors on input. *iam* is a process rank from [0, *nprocs*) interval. Each eigenvector resides entirely in one process. Each process holds a contiguous set of *nvs*(*iam*+2) eigenvectors. The global number of the first eigenvector that the process holds is: ((sum for *i*=[1, *iam*+1] of *nvs*(*i*))+1).

ldzi (local)

INTEGER. The leading dimension of the *zin* array.

iz, *jz* (global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER

Array of size *dlen_*. The array descriptor for the distributed matrix Z.

nvs (global) INTEGER.

Array of size *nprocs*+1

nvs(*i*) = number of eigenvectors held by processes [0, *i*-1)

nvs(1) = number of eigenvectors held by processes [0, 1 -1) = 0

nvs(*nprocs*+1)= number of eigenvectors held by processes [0, *nprocs*)= total number of eigenvectors.

key (global) INTEGER.

Array of size *n*. Indicates the actual index (after sorting) for each of the eigenvectors.

rwork (local).

REAL for pslaevswp

DOUBLE PRECISION for pdlaevswp

COMPLEX for pclaevswp

COMPLEX*16 for pzlaevswp.

Array of size *lrwork*.

lrwork (local)

INTEGER. Size of *work.*

Output Parameters

z (local).

REAL for pslaevswp

DOUBLE PRECISION for pdlaevswp

COMPLEX for pclaevswp

COMPLEX*16 for pzlaevswp.

Array of global size *n*by *n* and of local size (*lld_z*, *nq*). The eigenvectors on output. The eigenvectors are distributed in a block cyclic manner in both dimensions, with a block size of *nb*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lahrd

Reduces the first nb columns of a general rectangular matrix A so that elements below the k-th subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.

Syntax

```
call pslahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pdlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pclahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pzlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
```

Description

The p?lahrdroutine reduces the first *nb* columns of a real general *n*-by-(*n*-*k*+1) distributed matrix *A*(*ia*:*ia*+*n*-1 , *ja*:*ja*+*n*-*k*) so that elements below the *k*-th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation *Q'*A*Q*. The routine returns the matrices *V* and *T* which determine *Q* as a block reflector *I*-*V*T*V'*, and also the matrix *Y* = *A*V*T*.

This is an auxiliary routine called by p?gehrd. In the following comments sub(*A*) denotes *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

Input Parameters

n (global) INTEGER.

The order of the distributed matrix sub(*A*). *n* ≥ 0.

k (global) INTEGER.

The offset for the reduction. Elements below the *k*-th subdiagonal in the first *nb* columns are reduced to zero.

nb (global) INTEGER.

The number of columns to be reduced.

a (local).

REAL for pslahrd

DOUBLE PRECISION for pdlahrd

COMPLEX for pclahrd

COMPLEX*16 for pzlahrd.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*−*k*)). On entry, this array contains the local pieces of the *n*-by-(*n*−*k*+1) general distributed matrix *A*(*ia*:*ia*+*n*−1, *ja*:*ja*+*n*−*k*).

ia, *ja* (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

iy, *jy* (global) INTEGER. The row and column indices in the global matrix *Y* indicating the first row and the first column of the matrix sub(*Y*), respectively.

descy (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Y*.

work (local).

REAL for pslahrd

DOUBLE PRECISION for pdlahrd

COMPLEX for pclahrd

COMPLEX*16 for pzlahrd.

Array of size *nb*.

Output Parameters

a (local).

On exit, the elements on and above the *k*-th subdiagonal in the first *nb* columns are overwritten with the corresponding elements of the reduced distributed matrix; the elements below the *k*-th subdiagonal, with the array *tau*, represent the matrix *Q* as a product of elementary reflectors. The other columns of *A*(*ia*:*ia*+*n*−1, *ja*:*ja*+*n*−*k*) are unchanged. (See *Application Notes* below.)

tau (local)

REAL for pslahrd

DOUBLE PRECISION for pdlahrd

COMPLEX for pclahrd

COMPLEX*16 for pzlahrd.

Array of size *LOCc*(*ja*+*n*-2). The scalar factors of the elementary reflectors (see *Application Notes* below). *tau* is tied to the distributed matrix *A*.

t (local) REAL for pslahrd

`DOUBLE PRECISION` for `pdlahrd`

`COMPLEX` for `pclahrd`

`COMPLEX*16` for `pzlahrd`.

Array of size *nb_a*by *nb_a*. The upper triangular matrix *T*.

y (local).

`REAL` for `pslahrd`

`DOUBLE PRECISION` for `pdlahrd`

`COMPLEX` for `pclahrd`

`COMPLEX*16` for `pzlahrd`.

Pointer into the local memory to an array of size *lld_y*by *nb_a*. On exit, this array contains the local pieces of the *n*-by-*nb* distributed matrix *Y*. *lld_y* ≥ *LOCr*(`ia+n-1`).

Application Notes

The matrix *Q* is represented as a product of *nb* elementary reflectors

`Q = H(1)*H(2)*...*H(nb)`.

Each `H(i)` has the form

`H(i) = i-tau*v*v'`,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1: *i*+*k*-1)= 0, *v*(*i*+*k*)= 1; *v*(*i*+*k* +1:*n*) is stored on exit in *A*(*ia*+*i*+*k*:*ia*+*n*-1, *ja*+*i*-1), and *tau* in `tau`(`ja`+*i*-1).

The elements of the vectors *v* together form the (*n*-*k*+1)-by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form: `A(ia:ia+n-1`, *ja*:*ja*+*n*-*k*) := (*I*-*V***T***V'*)*(*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*)-*Y***V'*). The contents of `A(ia:ia+n-1`, *ja*:*ja*+*n*-*k*) on exit are illustrated by the following example with `n = 7`, `k = 3`, and `nb = 2`:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v1 & h & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

where *a* denotes an element of the original matrix `A(ia:ia+n-1, ja:ja+n-k)`, *h* denotes a modified element of the upper Hessenberg matrix *H*, and *vi* denotes an element of the vector defining `H(i)`.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laiect

Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).

Syntax

```
void pslaiect(float *sigma, int *n, float *d, int *count);
void pdlaiectb(float *sigma, int *n, float *d, int *count);
void pdlaiectl(float *sigma, int *n, float *d, int *count);
```

Description

The p?laiectroutine computes the number of negative eigenvalues of ($A - \sigma I$). This implementation of the Sturm Sequence loop exploits IEEE arithmetic and has no conditionals in the innermost loop. The signbit for real routine pslaiect is assumed to be bit 32. Double-precision routines pdlaiectb and pdlaiectl differ in the order of the double precision word storage and, consequently, in the signbit location. For pdlaiectb, the double precision word is stored in the big-endian word order and the signbit is assumed to be bit 32. For pdlaiectl, the double precision word is stored in the little-endian word order and the signbit is assumed to be bit 64.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

sigma

REALfor pslaiect

DOUBLE PRECISIONfor pdlaiectb/pdlaiectl.

The shift. p?laiect finds the number of eigenvalues less than equal to *sigma*.

n

INTEGER. The order of the tridiagonal matrix *T*. $n \geq 1$.

d

REALfor pslaiect

DOUBLE PRECISION for pdlaiectb/pdlaiectl.

Array of size $2n-1$.

On entry, this array contains the diagonals and the squares of the off-diagonal elements of the tridiagonal matrix *T*. These elements are assumed to be interleaved in memory for better cache performance. The diagonal entries of *T* are in the entries $d(1), d(3), \ldots, d(2n-1)$, while the squares of the off-diagonal entries are $d(2), d(4), \ldots, d(2n-2)$. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

Output Parameters

n

INTEGER. The count of the number of eigenvalues of *T* less than or equal to *sigma*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lamve

Copies all or part of one two-dimensional distributed array to another.

Syntax

```
call pslamve( uplo, m, n, a, ia, ja, desca, b, ib, jb, descb, dwork )
call pdlamve( uplo, m, n, a, ia, ja, desca, b, ib, jb, descb, dwork )
```

Description

`p?lamve` copies all or part of a distributed matrix *A* to another distributed matrix *B*. There is no alignment assumptions at all except that *A* and *B* are of the same size.

Input Parameters

uplo

(global) `CHARACTER*1`

Specifies the part of the distributed matrix sub(*A*) to be copied:

= 'U': Upper triangular part is copied; the strictly lower triangular part of sub(*A*) is not referenced;

= 'L': Lower triangular part is copied; the strictly upper triangular part of sub(*A*) is not referenced;

Otherwise: All of the matrix sub(*A*) is copied.

m

(global) `INTEGER`

The number of rows to be operated on, which is the number of rows of the distributed matrix sub(*A*). $m \geq 0$.

n

(global) `INTEGER`

The number of columns to be operated on, which is the number of columns of the distributed matrix sub(*A*). $n \geq 0$.

a

`REAL` for `pslamve`

`DOUBLE PRECISION` for `pdlamve`

(local) pointer into the local memory to an array of size (*lld_a*, $LOC_c(ja+n-1)$). This array contains the local pieces of the distributed matrix sub(*A*) to be copied from.

ia

(global) `INTEGER`

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja

(global) `INTEGER`

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

ib (global) INTEGER

The row index in the global matrix *B* indicating the first row of sub(*B*).

jb (global) INTEGER

The column index in the global matrix *B* indicating the first column of sub(*B*).

descb (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *B*.

dwork REAL for pslamve

DOUBLE PRECISION for pdlamve

(local workspace) array

If *uplo* = 'U' or *uplo* = 'L' and number of processors > 1, the length of *dwork* is at least as large as the length of *b*.

Otherwise, *dwork* is not referenced.

OUTPUT Parameters

b REAL for pslamve

DOUBLE PRECISION for pdlamve

(local) pointer into the local memory to an array of size (*lld_b*, LOC_c(*jb* +*n*-1)). This array contains on exit the local pieces of the distributed matrix sub(*B*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.

Syntax

```
val = pslange(norm, m, n, a, ia, ja, desca, work)
val = pdlange(norm, m, n, a, ia, ja, desca, work)
val = pclange(norm, m, n, a, ia, ja, desca, work)
val = pzlange(norm, m, n, a, ia, ja, desca, work)
```

Description

The p?langeroutine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

Input Parameters

norm

(global) CHARACTER. Specifies what value is returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*, it s not a matrix norm.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

m

(global) INTEGER.

The number of rows in the distributed matrix sub(A). When *m* = 0, p?lange is set to zero. $m \geq 0$.

n

(global) INTEGER.

The number of columns in the distributed matrix sub(*A*). When *n* = 0, p?lange is set to zero. $n \geq 0$.

a

(local).

REAL for pslange

DOUBLE PRECISION for pdlange

COMPLEX for pclange

COMPLEX*16 for pzlange.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

work

(local).

REAL for pslange

DOUBLE PRECISION for pdlange

COMPLEX for pclange

COMPLEX*16 for pzlange.

Array size *lwork*.

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),

*nq*0 if *norm* = '1', 'O' or 'o',

*mp*0 if *norm* = 'I' or 'i',

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool routines; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val

REAL for pslange/pclange

DOUBLE PRECISION for pdlange/pzlange

The value returned by the routine.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.

Syntax

val = pslanhs(*norm*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pdlanhs(*norm*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pclanhs(*norm*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pzlanhs(*norm*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

Description

The p?lanhsroutine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an upper Hessenberg distributed matrix sub(*A*) = *A*(*ia:ia+m-1*, *ja:ja+n-1*).

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

n (global) INTEGER.

The number of columns in the distributed matrix sub(*A*). When *n* = 0, p?lanhs is set to zero. *n*≥ 0.

a (local).

REAL for pslanhs

DOUBLE PRECISION for pdlanhs

COMPLEX for pclanhs

COMPLEX*16 for pzlanhs.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

work (local).

REAL for pslanhs

DOUBLE PRECISION for pdlanhs

COMPLEX for pclanhs

COMPLEX*16 for pzlanh.

Array of size *lwork*.

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),

*nq*0 if *norm* = '1', 'O' or 'o',

*mp*0 if *norm* = 'I' or 'i',

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool routines; *myrow*, *imycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val The value returned by the function.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lansy, p?lanhe

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric or a complex Hermitian matrix.

Syntax

```
val = pslansy(norm, uplo, n, a, ia, ja, desca, work)
val = pdlansy(norm, uplo, n, a, ia, ja, desca, work)
val = pclansy(norm, uplo, n, a, ia, ja, desca, work)
val = pzlansy(norm, uplo, n, a, ia, ja, desca, work)
val = pclanhe(norm, uplo, n, a, ia, ja, desca, work)
val = pzlanhe(norm, uplo, n, a, ia, ja, desca, work)
```

Description

The p?lansy and p?lanheroutines return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

Input Parameters

norm

(global) CHARACTER. Specifies what value is returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*, it s not a matrix norm.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo

(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is to be referenced.

= 'U': Upper triangular part of sub(*A*) is referenced,

= 'L': Lower triangular part of sub(*A*) is referenced.

n

(global) INTEGER.

The number of columns in the distributed matrix sub(*A*). When *n* = 0, p?lansy is set to zero. $n \geq 0$.

a

(local).

REAL for pslansy

DOUBLE PRECISION for pdlansy

COMPLEX for pclansy, pclanhe

COMPLEX*16 for pzlansy, pzlanhe.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix whose norm is to be computed, and the strictly lower triangular part of this matrix is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix whose norm is to be computed, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work

(local).

REAL for pslansy, pclansy, pclanhe

DOUBLE PRECISION for pdlansy, pzlansy, pzlanhe

Array of size *lwork*.

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),

2**nq*0+*mp*0+*ldw* if *norm* = '1', 'O' or 'o', 'I' or 'i',

where *ldw* is given by:

```
if( nprow≠npcol ) then
ldw = mb_a*iceil(iceil(np0,mb_a),(lcm/nprow))
else
ldw = 0
end if
```

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where *lcm* is the least common multiple of *nprow* and *npcol*, *lcm* = ilcm(*nprow*, *npcol*) and iceil(*x*,*y*) is a ScaLAPACK function that returns ceiling (*x*/*y*).

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),

ilcm, iceil, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val

The value returned by the routine.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lantr

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.

Syntax

```
val = pslantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pdlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pclantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pzlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
```

Description

The p?lantrroutine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

Input Parameters

norm

(global) CHARACTER. Specifies what value is returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*, it s not a matrix norm.

= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo

(global) CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is to be referenced.

= 'U': Upper trapezoidal,

= 'L': Lower trapezoidal.

Note that sub(*A*) is triangular instead of trapezoidal if *m* = *n*.

diag

(global) CHARACTER.

Specifies whether the distributed matrix sub(*A*) has unit diagonal.

= 'N': Non-unit diagonal.

= 'U': Unit diagonal.

m

(global) INTEGER.

The number of rows in the distributed matrix sub(*A*). When *m* = 0, p?lantr is set to zero. *m* ≥ 0.

n (global) INTEGER.

The number of columns in the distributed matrix sub(*A*). When *n* = 0, p?lantr is set to zero. *n* ≥ 0.

a (local).

REAL for pslantr

DOUBLE PRECISION for pdlantr

COMPLEX for pclantr

COMPLEX*16 for pzlantr.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

work (local).

REAL for pslantr

DOUBLE PRECISION for pdlantr

COMPLEX for pclantr

COMPLEX*16 for pzlantr.

Array size *lwork*.

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),

*nq*0 if *norm* = '1', 'O' or 'o',

*mp*0 if *norm* = 'I' or 'i',

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val The value returned by the routine.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lapiv

Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.

Syntax

```
call pslapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip, iwork)
call pdlapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip, iwork)
call pclapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip, iwork)
call pzlapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip, iwork)
```

Description

The p?lapivroutine applies either *P* (permutation matrix indicated by *ipiv*) or inv(*P*) to a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), resulting in row or column pivoting. The pivot vector may be distributed across a process row or a column. The pivot vector should be aligned with the distributed matrix *A*. This routine will transpose the pivot vector, if necessary.

For example, if the row pivots should be applied to the columns of sub(*A*), pass *rowcol*='C' and *pivroc*='C'.

Input Parameters

direc

(global) CHARACTER*1.

Specifies in which order the permutation is applied:

= 'F' (Forward): Applies pivots forward from top of matrix. Computes *P**sub(*A*).

= 'B' (Backward): Applies pivots backward from bottom of matrix. Computes inv(*P*)*sub(*A*).

rowcol

(global) CHARACTER*1.

Specifies if the rows or columns are to be permuted:

= 'R': Rows will be permuted,

= 'C': Columns will be permuted.

pivroc

(global) CHARACTER*1.

Specifies whether *ipiv* is distributed over a process row or column:

= 'R': *ipiv* is distributed over a process row,

= 'C': *ipiv* is distributed over a process column.

m

(global) INTEGER.

The number of rows in the distributed matrix sub(*A*). When *m* = 0, p?lapiv is set to zero. *m* ≥ 0.

n

(global) INTEGER.

The number of columns in the distributed matrix sub(*A*). When *n* = 0, p?lapiv is set to zero. *n* ≥ 0.

a

(local).

REAL for pslapiv

DOUBLE PRECISION for pdlapiv

COMPLEX for pclapiv

COMPLEX*16 for pzlapiv.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

ipiv (local) INTEGER.

Array of size *lipiv* ;

when *rowcol*='R' or 'r':

lipiv≥*LOCr*(*ia*+*m*-1) + *mb_a* if *pivroc*='C' or 'c',

lipiv≥*LOCc*(*m* + *mod*(*jp*-1, *nb_p*)) if *pivroc*='R' or 'r', and,

when *rowcol*='C' or 'c':

lipiv≥*LOCr*(*n* + mod(*ip*-1, *mb_p*)) if *pivroc*='C' or 'c',

lipiv≥*LOCc*(*ja*+*n*-1) + *nb_a* if *pivroc*='R' or 'r'.

This array contains the pivoting information. *ipiv(i)* is the global row (column), local row (column) *i* was swapped with. When *rowcol*='R' or 'r' and *pivroc*='C' or 'c', or *rowcol*='C' or 'c' and *pivroc*='R' or 'r', the last piece of this array of size *mb_a* (resp. *nb_a*) is used as workspace. In those cases, this array is tied to the distributed matrix *A*.

ip, *jp* (global) INTEGER. The row and column indices in the global matrix *P* indicating the first row and the first column of the matrix sub(*P*), respectively.

descip (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed vector *ipiv*.

iwork (local). INTEGER.

Array of size *ldw*, where *ldw* is equal to the workspace necessary for transposition, and the storage of the transposed *ipiv*:

Let *lcm* be the least common multiple of *nprow* and *npcol*.

```
if( rowcol.eq.'r' .and.  pivroc. eq.'r') then
      if( nprow.eq. npcol) then
ldw = LOCr( n_p + mod(jp-1, nb_p) ) + nb_p
     else
ldw = LOCr( n_p + mod(jp-1, nb_p) )+
nb_p * ceil( ceil(LOCc(n_p)/nb_p) / (lcm/npcol) )
   end if
 else if( rowcol.eq.'c' .and.  pivroc.eq.'c') then
```

```
       if( nprow.eq. npcol ) then
ldw = LOCc( m_p + mod(ip-1, mb_p) ) + mb_p
       else
ldw = LOCc( m_p + mod(ip-1, mb_p) ) +
mb_p *ceil(ceil(LOCr(m_p)/mb_p) / (lcm/nprow) )
       end if
 else
iwork is not referenced.
 end if
```

Output Parameters

a (local).

On exit, the local pieces of the permuted distributed submatrix.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lapv2

Applies a permutation to an m-by-n distributed matrix.

Syntax

```
call pslapv2 (direc, rowcol, m, n, a, ia, ja, desca, ipiv, ip, jp, descip )
call pdlapv2 (direc, rowcol, m, n, a, ia, ja, desca, ipiv, ip, jp, descip )
call pclapv2 (direc, rowcol, m, n, a, ia, ja, desca, ipiv, ip, jp, descip )
call pzlapv2 (direc, rowcol, m, n, a, ia, ja, desca, ipiv, ip, jp, descip )
```

Description

p?lapv2 applies either *P* (permutation matrix indicated by *ipiv*) or inv(*P*) to an *m*-by-*n* distributed matrix sub(*A*) denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), resulting in row or column pivoting. The pivot vector should be aligned with the distributed matrix *A*. For pivoting the rows of sub(*A*), *ipiv* should be distributed along a process column and replicated over all process rows. Similarly, *ipiv* should be distributed along a process row and replicated over all process columns for column pivoting.

Input Parameters

direc (global)

CHARACTER.

Specifies in which order the permutation is applied:

= 'F' (Forward) Applies pivots Forward from top of matrix. Computes *P* * sub(*A*);

= 'B' (Backward) Applies pivots Backward from bottom of matrix. Computes inv(*P*) * sub(*A*).

rowcol (global)

CHARACTER.

Specifies if the rows or columns are to be permuted:

= 'R' Rows will be permuted,

= 'C' Columns will be permuted.

m (global)

INTEGER.

The number of rows to be operated on, i.e. the number of rows of the distributed submatrix sub(*A*). *m* >= 0.

n (global)

INTEGER.

The number of columns to be operated on, i.e. the number of columns of the distributed submatrix sub(*A*). *n* >= 0.

a Pointer into local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)) .

On entry, this local array contains the local pieces of the distributed matrix sub(*A*) to which the row or columns interchanges will be applied.

ia (global)

INTEGER.

The row index in the global array *a* indicating the first row of sub(*A*).

ja (global)

INTEGER.

The column index in the global array *a* indicating the first column of sub(*A*).

desca (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

ipiv INTEGER.

Array, size >= LOCr(*m_a*)+*mb_a* if *rowcol* = 'R', LOCc(*n_a*)+*nb_a* otherwise.

It contains the pivoting information. *ipiv*(*i*) is the global row (column), local row (column) *i* was swapped with. The last piece of the array of size *mb_a* or *nb_a* is used as workspace. *ipiv* is tied to the distributed matrix *A*.

ip (global)

INTEGER.

The global row index of *ipiv*, which points to the beginning of the submatrix on which to operate.

jp (global)

INTEGER.

The global column index of *ipiv*, which points to the beginning of the submatrix on which to operate.

descip (global and local)

INTEGER.

Array of size 8.

The array descriptor for the distributed matrix *ipiv*.

Output Parameters

a — On exit, this array contains the local pieces of the permuted distributed matrix.

p?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ .

Syntax

```
call pslaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pdlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pclaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pzlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
```

Description

The p?laqgeroutine equilibrates a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) using the row and scaling factors in the vectors *r* and *c* computed by p?geequ.

Input Parameters

m — (global) INTEGER.

The number of rows in the distributed matrix sub(*A*). (*m* ≥0).

n — (global) INTEGER.

The number of columns in the distributed matrix sub(*A*). (*n* ≥0).

a — (local).

REAL for pslaqge

DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the distributed matrix sub(*A*).

ia, *ja* — (global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

r — (local).

REAL for pslaqge

DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

Array of size *LOCr(m_a)*. The row scale factors for sub(*A*). *r* is aligned with the distributed matrix *A*, and replicated across every process column. *r* is tied to the distributed matrix *A*.

c (local).

REAL for pslaqge

DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

Array of size *LOCc(n_a)*. The row scale factors for sub(*A*). *c* is aligned with the distributed matrix *A*, and replicated across every process column. *c* is tied to the distributed matrix *A*.

rowcnd (local).

REAL for pslaqge

DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

The global ratio of the smallest *r(i)* to the largest *r(i)*, $ia \le i \le ia+m-1$.

colcnd (local).

REAL for pslaqge

DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

The global ratio of the smallest *c(i)* to the largest *c(i)*, $ia \le i \le ia+n-1$.

amax (global). REAL for pslaqge

DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

Absolute value of largest distributed submatrix entry.

Output Parameters

a (local).

On exit, the equilibrated distributed matrix. See *equed* for the form of the equilibrated distributed submatrix.

equed (global) CHARACTER.

Specifies the form of equilibration that was done.

= 'N': No equilibration

= 'R': Row equilibration, that is, sub(*A*) has been pre-multiplied by diag(r(ia:ia+m-1)),

= 'C': column equilibration, that is, sub(*A*) has been post-multiplied by diag(c(ja:ja+n-1)),

= 'B': Both row and column equilibration, that is, sub(*A*) has been replaced by diag(r(ia:ia+m-1))* sub(*A*) * diag(c(ja:ja+n-1)).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laqr0

Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.

Syntax

call pslaqr0(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *h*, *desch*, *wr*, *wi*, *iloz*, *ihiz*, *z*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *info*, *reclevel*)

call pdlaqr0(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *h*, *desch*, *wr*, *wi*, *iloz*, *ihiz*, *z*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *info*, *reclevel*)

Description

p?laqr0 computes the eigenvalues of a Hessenberg matrix *H* and, optionally, the matrices T and *Z* from the Schur decomposition $H = Z*T*Z^T$, where *T* is an upper quasi-triangular matrix (the Schur form), and *Z* is the orthogonal matrix of Schur vectors.

Optionally *Z* may be postmultiplied into an input orthogonal matrix *Q* so that this routine can give the Schur factorization of a matrix *A* which has been reduced to the Hessenberg form *H* by the orthogonal matrix *Q*: $A = Q * H * Q^T = (QZ) * T * (QZ)^T$.

Input Parameters

wantt (global) LOGICAL

= .TRUE. : the full Schur form *T* is required;

= .FALSE. : only eigenvalues are required.

wantz (global) LOGICAL

= .TRUE. : the matrix of Schur vectors *Z* is required;

= .FALSE.: Schur vectors are not required.

n (global) INTEGER

The order of the Hessenberg matrix *H* (and *Z* if *wantz*). $n \geq 0$.

ilo, *ihi* (global) INTEGER

It is assumed that the matrix *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*. *ilo* and *ihi* are normally set by a previous call to p?gebal, and then passed to p?gehrd when the matrix output by *ihi* is reduced to Hessenberg form. Otherwise *ilo* and *ihi* should be set to 1 and *n*, respectively. If $n > 0$, then $1 \le ilo \le ihi \le n$.

If $n = 0$, then $ilo = 1$ and $ihi = 0$.

h

REAL for pslaqr0

DOUBLE PRECISION for pdlaqr0

(global) array of size (*lld_h*, $LOC_c(n)$)

The upper Hessenberg matrix *H*.

desch

(global and local) INTEGER

Array of size *dlen_*.

The array descriptor for the distributed matrix *H*.

iloz, *ihiz*

INTEGER

Specify the rows of the matrix *Z* to which transformations must be applied if *wantz* is .TRUE., $1 \le iloz \le ilo$; $ihi \le ihiz \le n$.

z

REAL for pslaqr0

DOUBLE PRECISION for pdlaqr0

Array of size (*lld_z*, $LOC_c(n)$).

If *wantz* is .TRUE., contains the matrix *Z*.

If *wantz*is .FALSE., *z* is not referenced.

descz

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *Z*.

work

REAL for pslaqr0

DOUBLE PRECISION for pdlaqr0

(local workspace) array of size *lwork*

lwork

(local) INTEGER

The length of the workspace array *work*.

iwork

(local workspace) INTEGER array of size *liwork*

liwork

(local) INTEGER

The length of the workspace array *iwork*.

reclevel

(local) INTEGER

Level of recursion. *reclevel* = 0 must hold on entry.

OUTPUT Parameters

h

On exit, if *wantt* is .TRUE., the matrix *H* is upper quasi-triangular in rows and columns *ilo*:*ihi*, with 1-by-1 and 2-by-2 blocks on the main diagonal. The 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with *H*(*i*,*i*) = *H*(*i*+1,*i*+1) and *H*(*i*+1,*i*)**H*(*i*,*i*+1) < 0. If *info* = 0 and *wantt*is .FALSE., the contents of *h* are unspecified on exit.

wr, *wi*

REAL for pslaqr0

DOUBLE PRECISION for pdlaqr0

The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)th, with *wi*(*i*) > 0 and *wi*(*i*+1) < 0. If *wantt* is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*.

z

Updated matrix with transformations applied only to the submatrix *Z*(*ilo*:*ihi*,*ilo*:*ihi*).

If COMPZ = 'I', on exit, if *info* = 0, *z* contains the orthogonal matrix *Z* of the Schur vectors of *H*.

If *wantz* is .TRUE., then *Z*(*ilo*:*ihi*,*iloz*:*ihi*z) is replaced by *Z*(*ilo*:*ihi*,*iloz*:*ihiz*)**U*, when *U* is the orthogonal/unitary Schur factor of *H*(*ilo*:*ihi*,*ilo*:*ihi*).

If *wantz*is .FALSE., then *z* is not defined.

work(1)

On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

iwork(1)

On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*.

info

INTEGER

> 0: if *info* = *i*, then the routine failed to compute all the eigenvalues. Elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

> 0: if *wantt*is .FALSE., then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *ihi* of the final output value of *h*.

> 0: if *wantt* is .TRUE., then (initial value of *H*)**U* = *U**(final value of *H*), where *U* is an orthogonal/unitary matrix. The final value of *H* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info*+1 through *ihi*.

> 0: if *wantz* is .TRUE., then (final value of *Z*(*ilo*:*ihi*,*iloz*:*ihiz*))=(initial value of *Z*(*ilo*:*ihi*,*iloz*:*ihiz*))**U*, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).

> 0: if *wantz*is .FALSE., then *z* is not accessed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laqr1

Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.

Syntax

```
call pslaqr1( wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work, lwork, iwork, ilwork, info )
```

```
call pdlaqr1( wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work, lwork, iwork, ilwork, info )
```

Description

`p?laqr1` is an auxiliary routine used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns *ilo* to *ihi*.

This is a modified version of `p?lahqr` from ScaLAPACK version 1.7.3. The following modifications were made:

- Workspace query functionality was added.
- Aggressive early deflation is implemented.
- Aggressive deflation (looking for two consecutive small subdiagonal elements by PSLACONSB) is abandoned.
- The returned Schur form is now in canonical form, i.e., the returned 2-by-2 blocks really correspond to complex conjugate pairs of eigenvalues.
- For some reason, the original version of `p?lahqr` sometimes did not read out the converged eigenvalues correctly. This is now fixed.

Input Parameters

wantt (global) `LOGICAL`

= `.TRUE.` : the full Schur form *T* is required;

= `.FALSE.`: only eigenvalues are required.

wantz (global) LOGICAL

= `.TRUE.` : the matrix of Schur vectors *Z* is required;

= `.FALSE.`: Schur vectors are not required.

n (global) `LOGICAL`

The order of the Hessenberg matrix *A* (and *Z* if *wantz*). $n \geq 0$.

ilo, *ihi* (global) `INTEGER`

It is assumed that the matrix *A* is already upper quasi-triangular in rows and columns *ihi*+1:*n*, and that A(*ilo*,*ilo*-1) = 0 (unless *ilo* = 1). p?laqr1 works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *H* if *wantt* is .TRUE..

1 ≤*ilo*≤ max(1,*ihi*); *ihi*≤*n*.

a

REAL for pslaqr1

DOUBLE PRECISION for pdlaqr1

(global) array of size (*lld_a*,LOC_c(*n*))

On entry, the upper Hessenberg matrix *A*.

desca

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

iloz, *ihiz*

(global) INTEGER

Specify the rows of the matrix *Z* to which transformations must be applied if *wantz* is .TRUE..

1 ≤*iloz*≤*ilo*; *ihi*≤*ihiz*≤*n*.

z

REAL for pslaqr1

DOUBLE PRECISION for pdlaqr1

(global) array of size (*lld_z*,LOC_c(*n*)).

If *wantz* is .TRUE., on entry *z* must contain the current matrix *Z* of transformations accumulated by p?hseqr

If *wantz* is .FALSE., *z* is not referenced.

descz

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *Z*.

work

REAL for pslaqr1

DOUBLE PRECISION for pdlaqr1

(local output) array of size *lwork*

lwork

(local) INTEGER

The size of the work array (*lwork*>=1).

If *lwork*=-1, then a workspace query is assumed.

iwork

(global and local) INTEGER array of size *ilwork*

This holds the some of the IBLK integer arrays.

ilwork

(local) INTEGER

The size of the *iwork* array (*ilwork*≥ 3).

OUTPUT Parameters

a	If *wantt* is .TRUE., the matrix *A* is upper quasi-triangular in rows and columns *ilo*:*ihi*, with any 2-by-2 or larger diagonal blocks not yet in standard form. If *wantt*is .FALSE., the contents of *a* are unspecified on exit.
wr, *wi*	REAL for pslaqr1 DOUBLE PRECISION for pdlaqr1 (global replicated) array of size *n* The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)th, with *wi*(*i*) > 0 and *wi*(*i*+1) < 0. If *wantt* is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *a*. *a* may be returned with larger diagonal blocks until the next release.
z	On exit *z* is updated; transformations are applied only to the submatrix *Z*(*iloz*:*ihiz*,*ilo*:*ihi*). If *wantz*is .FALSE., *z* is not referenced.
work(1)	On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.
info	(global) INTEGER < 0: parameter number -*info* incorrect or inconsistent = 0: successful exit > 0: p?laqr1 failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30*(*ihi*-*ilo*+1) iterations; if *info* = *i*, elements *i*+1:*ihi* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

Application Notes

This algorithm is very similar to p?ahqr. Unlike p?lahqr, instead of sending one double shift through the largest unreduced submatrix, this algorithm sends multiple double shifts and spaces them apart so that there can be parallelism across several processor row/columns. Another critical difference is that this algorithm aggregrates multiple transforms together in order to apply them in a block fashion.

Current Notes and/or Restrictions:

- This code requires the distributed block size to be square and at least six (6); unlike simpler codes like LU, this algorithm is extremely sensitive to block size. Unwise choices of too small a block size can lead to bad performance.
- This code requires *a* and *z* to be distributed identically and have identical contxts.
- This release currently does not have a routine for resolving the Schur blocks into regular 2x2 form after this code is completed. Because of this, a significant performance impact is required while the deflation is done by sometimes a single column of processors.
- This code does not currently block the initial transforms so that none of the rows or columns for any bulge are completed until all are started. To offset pipeline start-up it is recommended that at least 2*LCM(NPROW,NPCOL) bulges are used (if possible)
- The maximum number of bulges currently supported is fixed at 32. In future versions this will be limited only by the incoming *work* array.

- The matrix *A* must be in upper Hessenberg form. If elements below the subdiagonal are nonzero, the resulting transforms may be nonsimilar. This is also true with the LAPACK routine.
- For this release, it is assumed *rsrc_*=*csrc_*=0
- Currently, all the eigenvalues are distributed to all the nodes. Future releases will probably distribute the eigenvalues by the column partitioning.
- The internals of this routine are subject to change.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laqr2

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call pslaqr2( wantt, wantz, n, ktop, kbot, nw, a, desca, iloz, ihiz, z, descz, ns, nd,
sr, si, t, ldt, v, ldv, wr, wi, work, lwork )
```

```
call pdlaqr2( wantt, wantz, n, ktop, kbot, nw, a, desca, iloz, ihiz, z, descz, ns, nd,
sr, si, t, ldt, v, ldv, wr, wi, work, lwork )
```

Description

`p?laqr2` accepts as input an upper Hessenberg matrix *A* and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output *A*is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of *A*. It is to be hoped that the final version of *A* has many zero subdiagonal entries.

This routine handles small deflation windows which is affordable by one processor. Normally, it is called by `p?laqr1`. All the inputs are assumed to be valid without checking.

Input Parameters

wantt

(global) `LOGICAL`

If `.TRUE.`, then the Hessenberg matrix *A* is fully updated so that the quasi-triangular Schur factor may be computed (in cooperation with the calling subroutine).

If `.FALSE.`, then only enough of *A* is updated to preserve the eigenvalues.

wantz

(global) `LOGICAL`

If `.TRUE.`, then the orthogonal matrix *Z* is updated so that the orthogonal Schur factor may be computed (in cooperation with the calling subroutine).

If `.FALSE.`, then *z* is not referenced.

n

(global) `INTEGER`

The order of the matrix *A* and (if *wantz* is .TRUE.) the order of the orthogonal matrix *Z*.

ktop, *kbot*

(global) INTEGER

It is assumed without a check that either *kbot* = *n* or *A*(*kbot*+1,*kbot*)=0. *kbot* and *ktop* together determine an isolated block along the diagonal of the Hessenberg matrix. However, *A*(*ktop*,*ktop*-1)=0 is not essentially necessary if *wantt* is .TRUE. .

nw

(global) INTEGER

Deflation window size. 1 ≤*nw*≤ (*kbot*-*ktop*+1). Normally *nw*≥ 3 if p?laqr2 is called by p?laqr1.

a

REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(local) array of size (*lld_a*,LOC_c(*n*))

The initial *n*-by-*n* section of *a* stores the Hessenberg matrix undergoing aggressive early deflation.

desca

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

iloz, *ihiz*

(global) INTEGER

Specify the rows of *z* to which transformations must be applied if *wantz* is .TRUE.. 1 ≤*iloz*≤*ihiz*≤*n*.

z

REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

Array of size (*lld_z*,LOC_c(*n*))

If *wantz* is .TRUE., then on output, the orthogonal similarity transformation mentioned above has been accumulated into *z*(*iloz*:*ihiz*, *kbot*:*ktop*) from the right.

If *wantz* is .FALSE., then *z* is unreferenced.

descz

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *Z*.

t

REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(local workspace) array of size *ldt***nw*.

ldt

(local) INTEGER

The leading dimension of the array *t*. *ldt*≥*nw*.

v

REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(local workspace) array of size *ldv***nw*.

ldv (local) INTEGER

The leading dimension of the array *v*. *ldv*≥*nw*.

wr, *wi* REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(local workspace) array of size *kbot*.

work REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(local workspace) array of size *lwork*.

lwork (local) INTEGER

work(*lwork*) is a local array and *lwork* is assumed big enough so that *lwork*≥*nw***nw*.

OUTPUT Parameters

a On output *a* has been transformed by an orthogonal similarity transformation, perturbed, and returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

z

ns (global) INTEGER

The number of unconverged (that is, approximate) eigenvalues returned in *sr* and *si* that may be used as shifts by the calling subroutine.

nd (global) INTEGER

The number of converged eigenvalues uncovered by this subroutine.

sr, *si* REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(global) array of size *kbot*

On output, the real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in *sr*(*kbot*-*nd*-*ns*+1) through *sr*(*kbot*-*nd*) and *si*(*kbot*-*nd*-*ns*+1) through *si*(*kbot*-*nd*), respectively.

On processor #0, the real and imaginary parts of converged eigenvalues are stored in *sr*(*kbot*-*nd*+1) through *sr*(*kbot*) and *si*(*kbot*-*nd*+1) through *si*(*kbot*), respectively. On other processors, these entries are set to zero.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laqr3

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

call pslaqr3(*wantt*, *wantz*, *n*, *ktop*, *kbot*, *nw*, *h*, *desch*, *iloz*, *ihiz*, *z*, *descz*, *ns*, *nd*, *sr*, *si*, *v*, *descv*, *nh*, *t*, *desct*, *nv*, *wv*, *descw*, *work*, *lwork*, *iwork*, *liwork*, *reclevel*)

call pdlaqr3(*wantt*, *wantz*, *n*, *ktop*, *kbot*, *nw*, *h*, *desch*, *iloz*, *ihiz*, *z*, *descz*, *ns*, *nd*, *sr*, *si*, *v*, *descv*, *nh*, *t*, *desct*, *nv*, *wv*, *descw*, *work*, *lwork*, *iwork*, *liwork*, *reclevel*)

Description

This subroutine accepts as input an upper Hessenberg matrix *H* and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output *H* is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of *H*. It is to be hoped that the final version of *H* has many zero subdiagonal entries.

Input Parameters

wantt

(global) LOGICAL

If .TRUE., then the Hessenberg matrix *H* is fully updated so that the quasi-triangular Schur factor may be computed (in cooperation with the calling subroutine).

If .FALSE., then only enough of *H* is updated to preserve the eigenvalues.

wantz

(global) LOGICAL

If .TRUE., then the orthogonal matrix *Z* is updated so that the orthogonal Schur factor may be computed (in cooperation with the calling subroutine).

If .FALSE., then *z* is not referenced.

n

(global) INTEGER

The order of the matrix *H* and (if *wantz* is .TRUE.), the order of the orthogonal matrix *Z*.

ktop

(global) INTEGER

It is assumed that either *ktop* = 1 or *H* (*ktop*,*ktop*-1)=0. *kbot* and *ktop* together determine an isolated block along the diagonal of the Hessenberg matrix.

kbot

(global) INTEGER

It is assumed without a check that either *kbot* = *n* or *H* (*kbot*+1,*kbot*)=0. *kbot* and *ktop* together determine an isolated block along the diagonal of the Hessenberg matrix.

nw

(global) INTEGER

Deflation window size. 1 ≤*nw*≤ (*kbot*-*ktop*+1).

h

REAL for pslaqr3

DOUBLE PRECISION for pdlaqr3

(local) array of size (*lld_h*,$LOC_c(n)$)

The initial *n*-by-*n* section of *H* stores the Hessenberg matrix undergoing aggressive early deflation.

desch

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *H*.

iloz, *ihiz* (global) INTEGER

Specify the rows of the matrix *Z* to which transformations must be applied if *wantz* is .TRUE.. 1 ≤*iloz*≤*ihiz*≤*n*.

z REAL for pslaqr3

DOUBLE PRECISION for pdlaqr3

Array of size (*lld_z*,LOC_c(*n*))

If *wantz* is .TRUE., then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix *Z*(*iloz*:*ihiz*,*kbot*:*ktop*) from the right.

If *wantz* is .FALSE., then *z* is unreferenced.

descz (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *Z*.

v REAL for pslaqr3

DOUBLE PRECISION for pdlaqr3

(global workspace) array of size (*lld_v*,LOC_c(*nw*))

An *nw*-by-*nw* distributed work array.

descv (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *V*.

nh INTEGER scalar

The number of columns of *t*. *nh*≥*nw*.

t REAL for pslaqr3

DOUBLE PRECISION for pdlaqr3

(global workspace) array of size (*lld_t*,LOC_c(*nh*))

desct (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *T*.

nv (global) INTEGER

The number of rows of work array *wv* available for workspace. *nv*≥*nw*.

wv (global workspace) REAL array of size (*lld_w*,LOC_c(*nw*))

descw (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *wv*.

work (local workspace) REAL array of size *lwork*.

lwork (local) INTEGER

The size of the work array *work* (*lwork*≥1). *lwork* = 2**nw* suffices, but greater efficiency may result from larger values of *lwork*.

If *lwork* = -1, then a workspace query is assumed; p?laqr3 only estimates the optimal workspace size for the given values of *n*, *nw*, *ktop* and *kbot*. The estimate is returned in *work*(1). No error message related to *lwork* is issued by xerbla. Neither *h* nor *z* are accessed.

iwork (local workspace) INTEGER array of size *liwork*

liwork (local) INTEGER

The length of the workspace array *iwork* (*liwork*≥1).

If *liwork*=-1, then a workspace query is assumed.

OUTPUT Parameters

h On output *h* has been transformed by an orthogonal similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

z IF *wantz* is .TRUE., then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix *Z*(*iloz*:*ihiz*,*kbot*:*ktop*) from the right.

If *wantz* is .FALSE., then *z* is unreferenced.

ns (global) INTEGER

The number of unconverged (that is, approximate) eigenvalues returned in *sr* and *si* that may be used as shifts by the calling subroutine.

nd (global) INTEGER

The number of converged eigenvalues uncovered by this subroutine.

sr, *si* REAL for pslaqr3

DOUBLE PRECISION for pdlaqr3

(global) array of size *kbot*. The real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in *sr*(*kbot*-*nd*-*ns*+1) through *sr*(*kbot*-*nd*) and *si*(*kbot*-*nd*-*ns*+1) through *si*(*kbot*-*nd*), respectively. The real and imaginary parts of converged eigenvalues are stored in *sr*(*kbot*-*nd*+1) through *sr*(*kbot*) and *si*(*kbot*-*nd*+1) through *si*(*kbot*), respectively.

work(1) On exit, if *info* = 0, *work*(1) returns the optimal *lwork*

iwork(1) On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laqr4

Computes the eigenvalues of a Hessenberg matrix, and optionally computes the matrices from the Schur decomposition.

Syntax

call pslaqr4(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *a*, *desca*, *wr*, *wi*, *iloz*, *ihiz*, *z*, *descz*, *t*, *ldt*, *v*, *ldv*, *work*, *lwork*, *info*)

call pdlaqr4(*wantt*, *wantz*, *n*, *ilo*, *ihi*, *a*, *desca*, *wr*, *wi*, *iloz*, *ihiz*, *z*, *descz*, *t*, *ldt*, *v*, *ldv*, *work*, *lwork*, *info*)

Description

p?laqr4 is an auxiliary routine used to find the Schur decomposition and or eigenvalues of a matrix already in Hessenberg form from cols *ilo* to *ihi*. This routine requires that the active block is small enough, i.e. *ihi*-*ilo*+1 ≤*ldt*, so that it can be solved by LAPACK. Normally, it is called by p?laqr1. All the inputs are assumed to be valid without checking.

Input Parameters

wantt

(global) LOGICAL

= .TRUE.: the full Schur form *T* is required;

= .FALSE.: only eigenvalues are required.

wantz

(global) LOGICAL

= .TRUE.: the matrix of Schur vectors *Z* is required;

= .FALSE.: Schur vectors are not required.

n

(global) INTEGER

The order of the Hessenberg matrix *A* (and *Z* if *wantz*). *n*≥ 0.

ilo, *ihi*

(global) INTEGER

It is assumed that *a* is already upper quasi-triangular in rows and columns *ihi*+1:*n*, and that *A*(*ilo*,*ilo*-1) = 0 (unless *ilo* = 1). p?laqr4 works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *A* if *wantt* is .TRUE.. 1 ≤*ilo*≤ max(1,*ihi*); *ihi*≤*n*.

a

REAL for pslaqr4

DOUBLE PRECISION for pdlaqr4

(global) array of size (*lld_a*,$LOC_c(n)$)

The upper Hessenberg matrix *A*.

desca

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *a*.

iloz, *ihiz*

(global) INTEGER

Specify the rows of the matrix *Z* to which transformations must be applied if *wantz* is .TRUE.. 1 ≤*iloz*≤*ilo*; *ihi*≤*ihiz*≤*n*.

z

REAL for pslaqr4

DOUBLE PRECISION for pdlaqr4

(global) array.

If *wantz* is .TRUE., on entry *z* must contain the current matrix *Z* of transformations accumulated by p?hseqr.

If *wantz* is .FALSE., *z* is not referenced.

descz (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *Z*.

t REAL for pslaqr4

DOUBLE PRECISION for pdlaqr4

(local workspace) array of size *ldt**(*ihi*-*ilo*+1).

ldt (local) INTEGER

The leading dimension of the array *t*. *ldt*≥*ihi*-*ilo*+1.

v REAL for pslaqr4

DOUBLE PRECISION for pdlaqr4

(local workspace) array of size *ldv**(*ihi*-*ilo*+1).

ldv (local) INTEGER

The leading dimension of the array *v*. *ldv*≥*ihi*-*ilo*+1.

work REAL for pslaqr4

DOUBLE PRECISION for pdlaqr4

(local workspace) array of size *lwork*.

lwork (local) INTEGER

The size of the work array *work*.

lwork≥*ihi*-*ilo*+1.

OUTPUT Parameters

a On exit, if *wantt* is .TRUE., the matrix *A* is upper quasi-triangular in rows and columns *ilo*:*ihi*, with any 2-by-2 or larger diagonal blocks not yet in standard form. If *wantt*is .FALSE., the contents of *a* are unspecified on exit.

wr, *wi* REAL for pslaqr4

DOUBLE PRECISION for pdlaqr4

(global replicated) array of size *n*

The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)th, with *wi*(*i*) > 0 and *wi*(*i*+1) < 0. If *wantt* is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *a*. The matrix *A* may be returned with larger diagonal blocks until the next release.

z If *wantz* is .TRUE., *z* is updated with transformations applied only to the submatrix *Z*(*iloz*:*ihiz*,*ilo*:*ihi*).

info (global) INTEGER

< 0: parameter number -*info* incorrect or inconsistent;

= 0: successful exit;

> 0: p?laqr4 failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30*(*ihi*-*ilo*+1) iterations; if *info* = *i*, elements *i*+1:*ihi* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laqr5

Performs a single small-bulge multi-shift QR sweep.

Syntax

```
call pslaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, desch, iloz, ihiz, z, descz, work, lwork, iwork, liwork )
```

```
call pdlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, desch, iloz, ihiz, z, descz, work, lwork, iwork, liwork )
```

Description

This auxiliary subroutine called by p?laqr0 performs a single small-bulge multi-shift QR sweep by chasing separated groups of bulges along the main block diagonal of a Hessenberg matrix *H*.

Input Parameters

wantt (global) LOGICAL scalar

wantt= .TRUE. if the quasi-triangular Schur factor is being computed. *wantt* is set to .FALSE. otherwise.

wantz (global) LOGICAL scalar

wantz= .TRUE. if the orthogonal Schur factor is being computed. *wantz* is set to .FALSE. otherwise.

kacc22 (global) INTEGER

Value 0, 1, or 2. Specifies the computation mode of far-from-diagonal orthogonal updates.

= 0: p?laqr5 does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries.

= 1: p?laqr5 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.

= 2: p?laqr5 accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.

n (global) INTEGER scalar

	The order of the Hessenberg matrix *H* and, if *wantz*is .TRUE., the order of the orthogonal matrix *Z*.
ktop, *kbot*	(global) INTEGER scalar These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either *ktop* = 1 or *H*(*ktop*,*ktop*-1) = 0 and either *kbot* = *n* or *H*(*kbot*+1,*kbot*) = 0.
nshfts	(global) INTEGER scalar *nshfts* gives the number of simultaneous shifts. *nshfts* must be positive and even.
sr, *si*	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (global) Array of size *nshfts* *sr* contains the real parts and *si* contains the imaginary parts of the *nshfts* shifts of origin that define the multi-shift QR sweep.
h	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (local) Array of size (*lld_h*,$LOC_c(n)$) On input *h* contains a Hessenberg matrix *H*.
desch	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *H* .
iloz, *ihiz*	(global) INTEGER Specify the rows of the matrix *Z* to which transformations must be applied if *wantz*is .TRUE.. $1 \le iloz \le ihiz \le n$
z	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (local) array of size (*lld_z*,$LOC_c(n)$) If *wantz*= .TRUE., then the QR Sweep orthogonal similarity transformation is accumulated into the matrix *Z*(*iloz*:*ihiz*,*kbot*:*ktop*) from the right. If *wantz*= .FALSE., then *z* is unreferenced.
descz	(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*.
work	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (local workspace) array of size *lwork*
lwork	(local) INTEGER

	The size of the *work* array (*lwork*≥1). If *lwork*=-1, then a workspace query is assumed.
iwork	(local workspace) INTEGER array of size *liwork*
liwork	(local) INTEGER The size of the *iwork* array (*liwork*≥1). If *liwork*=-1, then a workspace query is assumed.

Output Parameters

h	A multi-shift QR sweep with shifts *sr*(*j*)+*i***si*(*j*) is applied to the isolated diagonal block in rows and columns *ktop* through *kbot* of the matrix *H*.
z	If *wantz*is .TRUE., *z* is updated with transformations applied only to the submatrix *Z*(*iloz*:*ihiz*,*kbot*:*ktop*).
work(1)	On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.
iwork(1)	On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ *.*

Syntax

```
call pslaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pdlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pclaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pzlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
```

Description

The p?laqsyroutine equilibrates a symmetric distributed matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) using the scaling factors in the vectors *sr* and *sc*. The scaling factors are computed by p?poequ.

Input Parameters

uplo	(global) CHARACTER. Specifies the upper or lower triangular part of the symmetric distributed matrix sub(*A*) is to be referenced: = 'U': Upper triangular part; = 'L': Lower triangular part.
n	(global) INTEGER. The order of the distributed matrix sub(*A*). *n* ≥ 0.
a	(local).

REAL for pslaqsy

DOUBLE PRECISION for pdlaqsy

COMPLEX for pclaqsy

COMPLEX*16 for pzlaqsy.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the distributed matrix sub(*A*). On entry, the local pieces of the distributed symmetric matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(*A*) is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

sr

(local)

REAL for pslaqsy

DOUBLE PRECISION for pdlaqsy

COMPLEX for pclaqsy

COMPLEX*16 for pzlaqsy.

Array of size *LOCr*(*m_a*). The scale factors for *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1). *sr* is aligned with the distributed matrix *A*, and replicated across every process column. *sr* is tied to the distributed matrix *A*.

sc

(local)

REAL for pslaqsy

DOUBLE PRECISION for pdlaqsy

COMPLEX for pclaqsy

COMPLEX*16 for pzlaqsy.

Array of size *LOCc*(*m_a*). The scale factors for *A* (*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1). *sc* is aligned with the distributed matrix *A*, and replicated across every process column. *sc* is tied to the distributed matrix *A*.

scond

(global). REAL for pslaqsy

DOUBLE PRECISION for pdlaqsy

COMPLEX for pclaqsy

COMPLEX*16 for pzlaqsy.

Ratio of the smallest *sr*(*i*) (respectively *sc*(*j*)) to the largest *sr*(*i*) (respectively *sc*(*j*)), with *ia* ≤ *i* ≤ *ia*+*n*-1 and *ja* ≤ *j* ≤ *ja*+*n*-1.

amax (global).

`REAL` for `pslaqsy`

`DOUBLE PRECISION` for `pdlaqsy`

`COMPLEX` for `pclaqsy`

`COMPLEX*16` for `pzlaqsy`.

Absolute value of largest distributed submatrix entry.

Output Parameters

a On exit,

if *equed* = `'Y'`, the equilibrated matrix:

`diag(`*sr*`(`*ia*`:`*ia*`+`*n*`-1)) * sub(`*A*`) * diag(`*sc*`(`*ja*`:`*ja*`+`*n*`-1))`.

equed (global)`CHARACTER*1`.

Specifies whether or not equilibration was done.

= `'N'`: No equilibration.

= `'Y'`: Equilibration was done, that is, sub(*A*) has been replaced by:

`diag(`*sr*`(`*ia*`:`*ia*`+`*n*`-1)) * sub(`*A*`) * diag(`*sc*`(`*ja*`:`*ja*`+`*n*`-1))`.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lared1d

Redistributes an array assuming that the input array, bycol, is distributed across rows and that all process columns contain the same copy of bycol.

Syntax

```
call pslared1d(n, ia, ja, desc, bycol, byall, work, lwork)
call pdlared1d(n, ia, ja, desc, bycol, byall, work, lwork)
```

Description

The `p?lared1d`routine redistributes a 1D array. It assumes that the input array *bycol* is distributed across rows and that all process column contain the same copy of *bycol*. The output array *byall* is identical on all processes and contains the entire array.

Input Parameters

np = Number of local rows in *bycol*()

n (global) `INTEGER`.

The number of elements to be redistributed. *n*≥ `0`.

ia, *ja* (global) `INTEGER`. *ia*, *ja* must be equal to 1.

desc (local) INTEGER array of size 9. A 2D array descriptor, which describes *bycol*.

bycol (local).

REAL for pslared1d

DOUBLE PRECISION for pdlared1d

COMPLEX for pclared1d

COMPLEX*16 for pzlared1d.

Distributed block cyclic array of global size *n* and of local size *np*. *bycol* is distributed across the process rows. All process columns are assumed to contain the same value.

work (local).

REAL for pslared1d

DOUBLE PRECISION for pdlared1d

COMPLEX for pclared1d

COMPLEX*16 for pzlared1d.

size *lwork*. Used to hold the buffers sent from one process to another.

lwork (local)

INTEGER. The size of the *work* array. *lwork* ≥ numroc(*n*, *desc*(*nb_*), 0, 0, *npcol*).

Output Parameters

byall (global). REAL for pslared1d

DOUBLE PRECISION for pdlared1d

COMPLEX for pclared1d

COMPLEX*16 for pzlared1d.

Global size *n*, local size *n*. *byall* is exactly duplicated on all processes. It contains the same values as *bycol*, but it is replicated across all processes rather than being distributed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lared2d

Redistributes an array assuming that the input array byrow is distributed across columns and that all process rows contain the same copy of byrow.

Syntax

```
call pslared2d(n, ia, ja, desc, byrow, byall, work, lwork)
call pdlared2d(n, ia, ja, desc, byrow, byall, work, lwork)
```

Description

The p?lared2droutine redistributes a 1D array. It assumes that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow*. The output array *byall* will be identical on all processes and will contain the entire array.

Input Parameters

np = Number of local rows in *byrow*()

n	(global) INTEGER. The number of elements to be redistributed. *n*≥ 0.
ia, *ja*	(global) INTEGER. *ia*, *ja* must be equal to 1.
desc	(local) INTEGER array of size *dlen_*. A 2D array descriptor, which describes *byrow*.
byrow	(local). REAL for pslared2d DOUBLE PRECISION for pdlared2d COMPLEX for pclared2d COMPLEX*16 for pzlared2d. Distributed block cyclic array of global size *n* and of local size *np*. *byrow* is distributed across the process columns. All process rows are assumed to contain the same value.
work	(local). REAL for pslared2d DOUBLE PRECISION for pdlared2d COMPLEX for pclared2d COMPLEX*16 for pzlared2d. size *lwork*. Used to hold the buffers sent from one process to another.
lwork	(local) INTEGER. The size of the *work* array. *lwork* ≥ numroc(*n*, *desc*(*nb_*), 0, 0, *npcol*).

Output Parameters

byall	(global). REAL for pslared2d DOUBLE PRECISION for pdlared2d COMPLEX for pclared2d COMPLEX*16 for pzlared2d. Global size *n*, local size *n*. *byall* is exactly duplicated on all processes. It contains the same values as *byrow*, but it is replicated across all processes rather than being distributed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call pslarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pdlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pclarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

The p?larfroutine applies a real/complex elementary reflector Q (or Q^T) to a real/complex m-by-n distributed matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

```
Q = I-tau*v*v',
```

where *tau* is a real/complex scalar and v is a real/complex vector.

If $tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

side (global). CHARACTER.

= 'L': form Q*sub(C),

= 'R': form sub(C)*Q, $Q=Q^T$.

m (global) INTEGER.

The number of rows in the distributed submatrix sub(A). $(m \geq 0)$.

n (global) INTEGER.

The number of columns in the distributed submatrix sub(A). $(n \geq 0)$.

v (local).

REAL for pslarf

DOUBLE PRECISION for pdlarf

COMPLEX for pclarf

COMPLEX*16 for pzlarf.

Pointer into the local memory to an array of size (*lld_v*,*), containing the local pieces of the global distributed matrix V representing the Householder transformation Q,

$V(iv:iv+m-1, jv)$ if $side$ = 'L' and $incv = 1$,

$V(iv, jv:jv+m-1)$ if $side$ = 'L' and $incv = m\_v$,

$V(iv:iv+n-1, jv)$ if $side$ = 'R' and $incv = 1$,

$V(iv, jv:jv+n-1)$ if $side$ = 'R' and $incv = m\_v$.

The array v is the representation of Q. v is not used if $tau = 0$.

iv, *jv*

(global) INTEGER. The row and column indices in the global matrix *V* indicating the first row and the first column of the matrix sub(*V*), respectively.

descv

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix V.

incv

(global) INTEGER.

The global increment for the elements of *V*. Only two values of *incv* are supported in this version, namely 1 and *m_v*.

incv must not be zero.

tau

(local).

REAL for pslarf

DOUBLE PRECISION for pdlarf

COMPLEX for pclarf

COMPLEX*16 for pzlarf.

Array of size *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

c

(local).

REAL for pslarf

DOUBLE PRECISION for pdlarf

COMPLEX for pclarf

COMPLEX*16 for pzlarf.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc* +*n*-1)), containing the local pieces of sub(*C*).

ic, *jc*

(global) INTEGER.

The row and column indices in the global matrix *C* indicating the first row and the first column of the matrix sub(*C*), respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local).

REAL for pslarf

DOUBLE PRECISION for pdlarf

COMPLEX for pclarf

COMPLEX*16 for pzlarf.

Array of size *lwork*.

If *incv* = 1,

if *side* = 'L',

if *ivcol* = *iccol*,

lwork≥*nqc*0
else
lwork≥*mpc*0 + max(1, *nqc*0)
end if
else if *side* = 'R' ,
lwork≥*nqc*0 + max(max(1, *mpc*0), numroc(numroc(*n*+
icoffc,*nb_v*,0,0,*npcol*),*nb_v*,0,0,*lcmq*))
end if
else if *incv* = *m_v*,
if *side* = 'L',
lwork≥*mpc*0 + max(max(*1*, *nqc*0), numroc(
numroc(*m*+*iroffc*,*mb_v*,0,0,*nprow*),*mb_v*,0,0, *lcmp*))
else if *side* = 'R',
if *ivrow* = *icrow*,
lwork≥*mpc*0
else
lwork≥*nqc*0 + max(*1*, *mpc*0)
end if
end if
end if,

where *lcm* is the least common multiple of *nprow* and *npcol* and *lcm* = *ilcm*(*nprow*, *npcol*), *lcmp* = *lcm*/*nprow*, *lcmq* = *lcm*/*npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).

On exit, sub(*C*) is overwritten by the *Q**sub(*C*) if *side* = 'L',

or sub(*C*) * *Q* if *side* = 'R'.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

call pslarfb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

call pdlarfb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

call pclarfb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

call pzlarfb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

Description

The p?larfbroutine applies a real/complex block reflector Q or its transpose Q^T/conjugate transpose Q^H to a real/complex distributed m-by-n matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Input Parameters

side (global) CHARACTER.

if *side* = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Left;

if *side* = 'R': apply Q or $Q^T$for real flavors (Q^H for complex flavors) from the Right.

trans (global) CHARACTER.

if *trans* = 'N': no transpose, apply Q;

for real flavors, if *trans*='T': transpose, apply Q^T

for complex flavors, if *trans* = 'C': conjugate transpose, apply Q^H;

direct (global) CHARACTER. Indicates how Q is formed from a product of elementary reflectors.

if *direct* = 'F': $Q = H(1)*H(2)*...*H(k)$ (Forward)

if *direct* = 'B': $Q = H(k)*...*H(2)*H(1)$ (Backward)

storev (global) CHARACTER.

Indicates how the vectors that define the elementary reflectors are stored:

if *storev* = 'C': Columnwise

if *storev* = 'R': Rowwise.

m (global) INTEGER.

The number of rows in the distributed matrix sub(C). $(m \geq 0)$.

n (global) INTEGER.

The number of columns in the distributed matrix sub(C). $(n \geq 0)$.

k (global) INTEGER.

The order of the matrix T.

v (local).

REAL for pslarfb

DOUBLE PRECISION for pdlarfb

COMPLEX for pclarfb

COMPLEX*16 for pzlarfb.

Pointer into the local memory to an array of size

(*lld_v*, *LOCc*(*jv*+*k*-1)) if *storev* = 'C',

(*lld_v*, *LOCc*(*jv*+*m*-1)) if *storev* = 'R' and *side* = 'L',

(*lld_v*, *LOCc*(*jv*+*n*-1)) if *storev* = 'R' and *side* = 'R'.

It contains the local pieces of the distributed vectors *V* representing the Householder transformation.

if *storev* = 'C' and *side* = 'L', *lld_v* ≥ max(1,*LOCr*(*iv*+*m*-1));

if *storev* = 'C' and *side* = 'R', *lld_v* ≥ max(1,*LOCr*(*iv*+*n*-1));

if *storev* = 'R', *lld_v*≥*LOCr*(*jv*+*k*-1).

iv, *jv* (global) INTEGER.

The row and column indices in the global matrix *V* indicating the first row and the first column of the matrix sub(*V*), respectively.

descv (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *V*.

c (local).

REAL for pslarfb

DOUBLE PRECISION for pdlarfb

COMPLEX for pclarfb

COMPLEX*16 for pzlarfb.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc* +*n*-1)), containing the local pieces of sub(*C*).

ic, *jc* (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the matrix sub(*C*), respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local).

REAL for pslarfb

DOUBLE PRECISION for pdlarfb

COMPLEX for pclarfb

COMPLEX*16 for pzlarfb.

Workspace array of size *lwork*.

If *storev* = 'C',

if *side* = 'L',

lwork≥ (*nqc*0 + *mpc*0) * *k*

else if *side* = 'R',

lwork ≥ (*nqc*0 + max(*npv*0 + numroc(numroc(*n* +

icoffc, *nb_v*, 0, 0, *npcol*), *nb_v*, 0, 0, *lcmq*),

*mpc*0)) * *k*

end if

else if *storev* = 'R' ,

if *side* = 'L' ,

lwork≥ (*mpc*0 + max(*mqv*0 + numroc(numroc(*m* +

iroffc, *mb_v*, 0, 0, *nprow*), *mb_v*, 0, 0, *lcmp*),

*nqc*0)) * *k*

else if *side* = 'R',

lwork ≥ (*mpc*0 + *nqc*0) * *k*

end if

end if,

where

lcmq = *lcm* / *npcol* with *lcm* = iclm(*nprow*, *npcol*),

iroffv = mod(*iv*-1, *mb_v*), *icoffv* = mod(*jv*-1, *nb_v*),

ivrow = indxg2p(*iv*, *mb_v*, *myrow*, *rsrc_v*, *nprow*),

ivcol = indxg2p(*jv*, *nb_v*, *mycol*, *csrc_v*, *npcol*),

MqV0 = numroc(*m*+*icoffv*, *nb_v*, *mycol*, *ivcol*, *npcol*),

NpV0 = numroc(*n*+*iroffv*, *mb_v*, *myrow*, *ivrow*, *nprow*),

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc*_c, *nprow*),

iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc*_c, *npcol*),

MpC0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

NpC0 = numroc(*n*+*icoffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

NqC0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

t (local).

`REAL` for `pslarfb`

`DOUBLE PRECISION` for `pdlarfb`

`COMPLEX` for `pclarfb`

`COMPLEX*16` for `pzlarfb`.

Array of size `( mb_v, mb_v)` if `storev = 'R'`, and `( nb_v, nb_v)` if `storev = 'C'`. The triangular matrix *t* is the representation of the block reflector.

c (local).

On exit, sub(*C*) is overwritten by the `Q*sub(C)`, or `Q'*sub(C)`, or `sub(C)*Q`, or `sub(C)*Q'`. *Q*′ is transpose (conjugate transpose) of *Q*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larfc

Applies the conjugate transpose of an elementary reflector to a general matrix.

Syntax

```
call pclarfc(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarfc(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

The `p?larfc`routine applies a complex elementary reflector Q^H to a complex *m*-by-*n* distributed matrix sub(*C*) = `C(ic:ic+m-1, jc:jc+n-1)`, from either the left or the right. *Q* is represented in the form

```
Q = i-tau*v*v',
```

where *tau* is a complex scalar and *v* is a complex vector.

If `tau = 0`, then *Q* is taken to be the unit matrix.

Input Parameters

side (global) `CHARACTER`.

if `side = 'L'`: form Q^H`*sub(C)` ;

if `side = 'R'`: form sub `(C)*`Q^H.

m (global) `INTEGER`.

The number of rows in the distributed matrix sub(*C*). `(m ≥ 0)`.

n (global) `INTEGER`.

The number of columns in the distributed matrix sub(*C*). `(n ≥ 0)`.

v (local).

`COMPLEX` for `pclarfc`

COMPLEX*16 for pzlarfc.

Pointer into the local memory to an array of size (*lld_v*,*), containing the local pieces of the global distributed matrix *V* representing the Householder transformation *Q*,

V(*iv*:*iv*+*m*-1, *jv*) if *side* = 'L' and *incv* = 1,

V(*iv*, *jv*:*jv*+*m*-1) if *side* = 'L' and *incv* = *m_v*,

V(*iv*:*iv*+*n*-1, *jv*) if *side* = 'R' and *incv* = 1,

V(*iv*, *jv*:*jv*+*n*-1) if *side* = 'R' and *incv* = *m_v*.

The array *v* is the representation of *Q*. *v* is not used if *tau* = 0.

iv, *jv* (global) INTEGER.

The row and column indices in the global matrix *V* indicating the first row and the first column of the matrix sub(*V*), respectively.

descv (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *V*.

incv (global) INTEGER.

The global increment for the elements of *v*. Only two values of *incv* are supported in this version, namely 1 and *m_v*.

incv must not be zero.

tau (local)

COMPLEX for pclarfc

COMPLEX*16 for pzlarfc.

Array of size *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

c (local).

COMPLEX for pclarfc

COMPLEX*16 for pzlarfc.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic, *jc* (global) INTEGER.

The row and column indices in the global matrix *C* indicating the first row and the first column of the matrix sub(*C*), respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local).

COMPLEX for pclarfc

COMPLEX*16 for pzlarfc.

Workspace array of size *lwork*.

If *incv* = 1,
 if *side* = 'L' ,
  if *ivcol* = *iccol*,
   lwork ≥ *nqc*0
  else
   lwork ≥ *mpc*0 + max(1, *nqc*0)
  end if
 else if *side* = 'R',
  lwork ≥ *nqc*0 + max(max(1, *mpc*0), numroc(numroc(
   n+*icoffc*,*nb_v*,0,0,*npcol*), *nb_v*,0,0,*lcmq*))
 end if
else if *incv* = *m_v*,
 if *side* = 'L',
  lwork ≥ *mpc*0 + max(max(1, *nqc*0), numroc(numroc(
   m+*iroffc*,*mb_v*,0,0,*nprow*),*mb_v*,0,0,*lcmp*))
 else if *side* = 'R' ,
  if *ivrow* = *icrow*,
   lwork ≥ *mpc*0
  else
   lwork ≥ *nqc*0 + max(1, *mpc*0)
  end if
 end if
end if,

where *lcm* is the least common multiple of *nprow* and *npcol* and *lcm* = ilcm(*nprow*, *npcol*),

lcmp = *lcm*/*nprow*, *lcmq* = *lcm*/*npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions;*myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).

On exit, sub(*C*) is overwritten by the Q^H*sub(*C*) if *side* = 'L', or sub(*C*) * Q^H if *side* = 'R'.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call pslarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pdlarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pclarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pzlarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
```

Description

The p?larfgroutine generates a real/complex elementary reflector *H* of order *n*, such that

$$H * \mathrm{sub}(X) = H * \begin{pmatrix} x(iax, jax) \\ x \end{pmatrix} = \begin{pmatrix} alpha \\ 0 \end{pmatrix}, \quad H' * H = I,$$

where *alpha* is a scalar (a real scalar - for complex flavors), and sub(*X*) is an (*n*-1)-element real/complex distributed vector *X*(*ix*:*ix*+*n*-2, *jx*) if *incx* = 1 and *X*(*ix*, *jx*:*jx*+*n*-2) if *incx* = *m_x*. *H* is represented in the form

$$H = I - tau * \begin{pmatrix} 1 \\ v \end{pmatrix} * (1 \; v')$$

where *tau* is a real/complex scalar and *v* is a real/complex (*n*-1)-element vector. Note that *H* is not Hermitian.

If the elements of sub(*X*) are all zero (and *X*(*iax*, *jax*) is real for complex flavors), then *tau* = 0 and *H* is taken to be the unit matrix.

Otherwise 1 ≤ real(*tau*) ≤ 2 and abs(*tau*-1) ≤ 1.

Input Parameters

n — (global) INTEGER.

The global order of the elementary reflector. *n* ≥ 0.

iax, *jax* — (global) INTEGER.

The global row and column indices of *X*(*iax*, *jax*) in the global matrix *X*.

x (local).

REAL for pslarfg

DOUBLE PRECISION for pdlarfg

COMPLEX for pclarfg

COMPLEX*16 for pzlarfg.

Pointer into the local memory to an array of size (*lld_x*, *). This array contains the local pieces of the distributed vector sub(*X*). Before entry, the incremented array sub(*X*) must contain vector *x*.

ix, *jx* (global) INTEGER.

The row and column indices in the global matrix *X* indicating the first row and the first column of sub(*X*), respectively.

descx (global and local) INTEGER.

Array of size *dlen_*. The array descriptor for the distributed matrix *X*.

incx (global) INTEGER.

The global increment for the elements of *x*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

alpha (local)

REAL for pslafg

DOUBLE PRECISION for pdlafg

COMPLEX for pclafg

COMPLEX*16 for pzlafg.

On exit, *alpha* is computed in the process scope having the vector sub(*X*).

x (local).

On exit, it is overwritten with the vector *v*.

tau (local).

REAL for pslarfg

DOUBLE PRECISION for pdlarfg

COMPLEX for pclarfg

COMPLEX*16 for pzlarfg.

Array of size *LOCc*(*jx*) if *incx* = 1, and *LOCr*(*ix*) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *X*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larft

Forms the triangular vector `T` *of a block reflector* `H=I-V*T*V^H`.

Syntax

```
call pslarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pdlarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pclarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pzlarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

Description

The `p?larft`routine forms the triangular factor *T* of a real/complex block reflector *H* of order *n*, which is defined as a product of *k* elementary reflectors.

If `direct = 'F'`, `H = H(1)*H(2)...*H(k)`, and *T* is upper triangular;

If `direct = 'B'`, `H = H(k)*...*H(2)*H(1)`, and *T* is lower triangular.

If `storev = 'C'`, the vector which defines the elementary reflector `H(i)` is stored in the *i*-th column of the distributed matrix *V*, and

`H = I-V*T*V'`

If `storev = 'R'`, the vector which defines the elementary reflector `H(i)` is stored in the *i*-th row of the distributed matrix *V*, and

`H = I-V'*T*V`.

Input Parameters

direct

(global) `CHARACTER*1`.

Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

if `direct = 'F'`: `H = H(1)*H(2)*...*H(k)` (forward)

if `direct = 'B'`: `H = H(k)*...*H(2)*H(1)` (backward).

storev

(global) `CHARACTER*1`.

Specifies how the vectors that define the elementary reflectors are stored (See *Application Notes* below):

if `storev = 'C'`: columnwise;

if `storev = 'R'`: rowwise.

n

(global) `INTEGER`.

The order of the block reflector *H*. $n \geq 0$.

k

(global) `INTEGER`.

The order of the triangular factor *T*, is equal to the number of elementary reflectors.

$1 \leq k \leq mb\_v$ (= nb_v).

v

`REAL` for `pslarft`

DOUBLE PRECISION for pdlarft

COMPLEX for pclarft

COMPLEX*16 for pzlarft.

Pointer into the local memory to an array of local size

(*LOCr*(*iv*+*n*-1), *LOCc*(*jv*+*k*-1)) if *storev* = 'C', and

(*LOCr*(*iv*+*k*-1), *LOCc*(*jv*+*n*-1)) if *storev* = 'R'.

The distributed matrix *V* contains the Householder vectors. (See *Application Notes* below).

iv, *jv*

(global) INTEGER.

The row and column indices in the global matrix *V* indicating the first row and the first column of the matrix sub(*V*), respectively.

descv

(local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *V*.

tau

(local)

REAL for pslarft

DOUBLE PRECISION for pdlarft

COMPLEX for pclarft

COMPLEX*16 for pzlarft.

Array of size *LOCr*(*iv*+*k*-1) if *incv* = *m_v*, and *LOCc*(*jv*+*k*-1) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

work

(local).

REAL for pslarft

DOUBLE PRECISION for pdlarft

COMPLEX for pclarft

COMPLEX*16 for pzlarft.

Workspace array of size *k**(*k* -1)/2.

Output Parameters

v

REAL for pslarft

DOUBLE PRECISION for pdlarft

COMPLEX for pclarft

COMPLEX*16 for pzlarft.

t

(local)

REAL for pslarft

DOUBLE PRECISION for pdlarft

COMPLEX for pclarft

COMPLEX*16 for pzlarft.

Array of size (*nb_v*, *nb_v*) if *storev* = 'C', and (*mb_v*, *mb_v*) otherwise. It contains the *k*-by-*k* triangular factor of the block reflector associated with *v*. If *direct* = 'F', *t* is upper triangular;

if *direct* = 'B', *t* is lower triangular.

Application Notes

The shape of the matrix *V* and the storage of the vectors that define the *H(i)* is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C':

$$V(iv:iv+n-1,\ jv:jv+k-1) = \begin{bmatrix} 1 & & \\ v1 & 1 & \\ v1 & v2 & 1 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}$$

direct = 'F' and *storev* = 'R'

$$V(iv:iv+k-1,\ jv:jv+n-1) = \begin{bmatrix} 1 & v1 & v1 & v1 \\ & 1 & v2 & v2 \\ & & 1 & v3 \end{bmatrix}$$

direct = 'B' and *storev* = 'C'

$$V(iv:iv+n-1,\ jv:jv+k-1) = \begin{bmatrix} v1 & v2v & v3 \\ v1 & v2 & v3 \\ 1 & v2 & v3 \\ & 1 & v3 \\ & & 1 \end{bmatrix}$$

direct = 'B' and *storev* = 'R'

$$V(iv:iv+k-1,\ jv:jv+n-1) = \begin{bmatrix} v1 & v1 & 1 & & \\ v2 & v2 & v2 & 1 & \\ v3 & v3 & v3 & v3 & 1 \end{bmatrix}$$

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larz

Applies an elementary reflector as returned by p?tzrzf *to a general matrix.*

Syntax

```
call pslarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pdlarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pclarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

The p?larzroutine applies a real/complex elementary reflector *Q* (or Q^T) to a real/complex *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1), from either the left or the right. *Q* is represented in the form

`Q = I-tau*v*v'`,

where *tau* is a real/complex scalar and *v* is a real/complex vector.

If `tau = 0`, then *Q* is taken to be the unit matrix.

Q is a product of *k* elementary reflectors as returned by `p?tzrzf`.

Input Parameters

side (global) `CHARACTER`.

if `side = 'L'`: form `Q*sub(C)`,

if `side = 'R'`: form `sub(C)*Q`, `Q = Q^T` (for real flavors).

m (global) `INTEGER`.

The number of rows in the distributed matrix sub(*C*). `(m≥ 0)`.

n (global) `INTEGER`.

The number of columns in the distributed matrix sub(*C*). `(n≥ 0)`.

l (global) `INTEGER`.

The columns of the distributed matrix sub(*A*) containing the meaningful part of the Householder reflectors. If `side = 'L'`, `m≥ l≥` 0,

if `side = 'R'`, `n≥l ≥ 0`.

v (local).

`REAL` for `pslarz`

`DOUBLE PRECISION` for `pdlarz`

`COMPLEX` for `pclarz`

`COMPLEX*16` for `pzlarz`.

Pointer into the local memory to an array of size (*lld_v*,*) containing the local pieces of the global distributed matrix *V* representing the Householder transformation *Q*,

V`(iv:iv+l-1, jv)` if `side = 'L'` and `incv = 1`,

V`(iv, jv:jv+l-1)` if `side = 'L'` and `incv = m_v`,

V`(iv:iv+l-1, jv)` if `side = 'R'` and `incv = 1`,

V`(iv, jv:jv+l-1)` if `side = 'R'` and `incv = m_v`.

The vector `v` in the representation of *Q*. `v` is not used if `tau = 0`.

iv, *jv* (global) `INTEGER`. The row and column indices in the global distributed matrix *V* indicating the first row and the first column of the matrix sub(*V*), respectively.

descv (global and local) `INTEGER` array of size *dlen_*. The array descriptor for the distributed matrix *V*.

incv (global) `INTEGER`.

The global increment for the elements of *V*. Only two values of *incv* are supported in this version, namely 1 and *m_v*.

incv must not be zero.

tau (local)

REAL for pslarz

DOUBLE PRECISION for pdlarz

COMPLEX for pclarz

COMPLEX*16 for pzlarz.

Array of size *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

c (local).

REAL for pslarz

DOUBLE PRECISION for pdlarz

COMPLEX for pclarz

COMPLEX*16 for pzlarz.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic, *jc* (global) INTEGER.

The row and column indices in the global matrix *C* indicating the first row and the first column of the matrix sub(*C*), respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local).

REAL for pslarz

DOUBLE PRECISION for pdlarz

COMPLEX for pclarz

COMPLEX*16 for pzlarz.

Array of size *lwork*

If *incv* = 1,

if *side* = 'L' ,

if *ivcol* = *iccol*,

lwork ≥ *NqC*0

else

lwork ≥ *MpC*0 + max(1, *NqC*0)

end if

else if *side* = 'R' ,

lwork ≥ *NqC*0 + max(max(1, *MpC*0), numroc(numroc(*n* +*icoffc*,*nb_v*,0,0,*npcol*),*nb_v*,0,0,*lcmq*))

end if

else if *incv* = *m_v*,

if *side* = 'L' ,

lwork ≥ *MpC*0 + max(max(1, *NqC*0), numroc(numroc(*m*+*iroffc*,*mb_v*,0,0,*nprow*),*mb_v*,0,0,*lcmp*))

else if *side* = 'R' ,

if *ivrow* = *icrow*,

lwork ≥ *MpC*0

else

lwork ≥ *NqC*0 + max(1, *MpC*0)

end if

end if

end if.

Here *lcm* is the least common multiple of *nprow* and *npcol* and

lcm = ilcm(*nprow*, *npcol*), *lcmp* = *lcm* / *nprow*,

lcmq = *lcm* / *npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).

On exit, sub(*C*) is overwritten by the *Q**sub(*C*) if *side* = 'L', or sub(*C*)**Q* if *side* = 'R'.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larzb

Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzrzf *to a general matrix.*

Syntax

call pslarzb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *l*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

```
call pdlarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc, descc, work)
call pclarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc, descc, work)
call pzlarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc, descc, work)
```

Description

The p?larzbroutine applies a real/complex block reflector Q or its transpose Q^T (conjugate transpose Q^H for complex flavors) to a real/complex distributed m-by-n matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Q is a product of k elementary reflectors as returned by p?tzrzf.

Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

side (global) CHARACTER.

if *side* = 'L': apply Q or Q^T (Q^H for complex flavors) from the Left;

if *side* = 'R': apply Q or Q^T (Q^H for complex flavors) from the Right.

trans (global) CHARACTER.

if *trans* = 'N': No transpose, apply Q;

If *trans*='T': Transpose, apply Q^T (real flavors);

If *trans*='C': Conjugate transpose, apply Q^H (complex flavors).

direct (global) CHARACTER.

Indicates how H is formed from a product of elementary reflectors.

if *direct* = 'F': $H = H(1)*H(2)*...*H(k)$ - forward (not supported) ;

if *direct* = 'B': $H = H(k)*...*H(2)*H(1)$ - backward.

storev (global) CHARACTER.

Indicates how the vectors that define the elementary reflectors are stored:

if *storev* = 'C': columnwise (not supported).

if *storev* = 'R': rowwise.

m (global) INTEGER.

The number of rows in the distributed submatrix sub(C). $(m \geq 0)$.

n (global) INTEGER.

The number of columns in the distributed submatrix sub(C). $(n \geq 0)$.

k (global) INTEGER.

The order of the matrix T. (= the number of elementary reflectors whose product defines the block reflector).

l (global) INTEGER.

The columns of the distributed submatrix sub(*A*) containing the meaningful part of the Householder reflectors.

If *side* = 'L', $m \geq l \geq 0$,

if *side* = 'R', $n \geq l \geq 0$.

v (local).

REAL for pslarzb

DOUBLE PRECISION for pdlarzb

COMPLEX for pclarzb

COMPLEX*16 for pzlarzb.

Pointer into the local memory to an array of size (*lld_v*, *LOCc*(*jv*+*m*-1)) if *side* = 'L', (*lld_v*, *LOCc*(*jv*+*m*-1)) if *side* = 'R'.

It contains the local pieces of the distributed vectors *V* representing the Householder transformation as returned by p?tzrzf.

lld_v ≥ *LOCr*(*iv*+*k*-1).

iv, *jv* (global) INTEGER.

The row and column indices in the global matrix *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *V*.

t (local)

REAL for pslarzb

DOUBLE PRECISION for pdlarzb

COMPLEX for pclarzb

COMPLEX*16 for pzlarzb.

Array of size *mb_v*by *mb_v*.

The lower triangular matrix *T* in the representation of the block reflector.

c (local).

REAL for pslarfb

DOUBLE PRECISION for pdlarfb

COMPLEX for pclarfb

COMPLEX*16 for pzlarfb.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc*+*n*-1)).

On entry, the *m*-by-*n* distributed matrix sub(*C*).

ic, *jc* (global) INTEGER.

The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local).

REAL for pslarzb

DOUBLE PRECISION for pdlarzb

COMPLEX for pclarzb

COMPLEX*16 for pzlarzb.

Array of size *lwork*.

```
If storev = 'C' ,
  if side = 'L' ,
    lwork≥(nqc0 + mpc0)* k
  else if side = 'R' ,
    lwork ≥ (nqc0 + max(npv0 + numroc(numroc(n+icoffc, nb_v, 0, 0,
npcol),
           nb_v, 0, 0, lcmq), mpc0))* k
  end if
else if storev = 'R' ,
  if side = 'L' ,
    lwork≥ (mpc0 + max(mqv0 + numroc(numroc(m+iroffc, mb_v, 0, 0,
nprow),
             mb_v, 0, 0, lcmp), nqc0))* k
  else if side = 'R' ,
    lwork≥ (mpc0 + nqc0) * k
  end if
  end if.
```

Here *lcmq* = *lcm*/*npcol* with *lcm* = iclm(*nprow*, *npcol*),

iroffv = mod(*iv*-1, *mb_v*), *icoffv* = mod(*jv-1*, *nb_v*),

ivrow = indxg2p(*iv*, *mb_v*, *myrow*, *rsrc_v*, *nprow*),

ivcol = indxg2p(*jv*, *nb_v*, *mycol*, *csrc_v*, *npcol*),

mqv0 = numroc(*m*+*icoffv*, *nb_v*, *mycol*, *ivcol*, *npcol*),

npv0 = numroc(*n*+*iroffv*, *mb_v*, *myrow*, *ivrow*, *nprow*),

iroffc = mod(*ic*-1, *mb_c*), *icoffc*= *mod*(*jc*-1, *nb_c*),

icrow= indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol= indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

mpc0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

npc0 = numroc(*n*+*icoffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

nqc0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).

On exit, sub(*C*) is overwritten by the *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*, or sub(*C*)**Q*', where *Q*' is the transpose (conjugate transpose) of *Q*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larzc

Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzrzf *to a general matrix.*

Syntax

call pclarzc(*side*, *m*, *n*, *l*, *v*, *iv*, *jv*, *descv*, *incv*, *tau*, *c*, *ic*, *jc*, *descc*, *work*)

call pzlarzc(*side*, *m*, *n*, *l*, *v*, *iv*, *jv*, *descv*, *incv*, *tau*, *c*, *ic*, *jc*, *descc*, *work*)

Description

The p?larzcroutine applies a complex elementary reflector Q^H to a complex *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1), from either the left or the right. *Q* is represented in the form

Q = *i*-*tau***v***v*',

where *tau* is a complex scalar and *v* is a complex vector.

If *tau* = 0, then *Q* is taken to be the unit matrix.

Q is a product of *k* elementary reflectors as returned by p?tzrzf.

Input Parameters

side (global) CHARACTER.

if *side* = 'L': form Q^H*sub(*C*);

if *side* = 'R': form sub(*C*)*Q^H.

m (global) INTEGER.

The number of rows in the distributed matrix sub(*C*). (*m* ≥ 0).

n (global) INTEGER.

The number of columns in the distributed matrix sub(*C*). (*n* ≥ 0).

l (global) INTEGER.

The columns of the distributed matrix sub(*A*) containing the meaningful part of the Householder reflectors.

If *side* = 'L', $m \geq l \geq 0$,

if *side* = 'R', $n \geq l \geq 0$.

v (local).

COMPLEX for pclarzc

COMPLEX*16 for pzlarzc.

Pointer into the local memory to an array of size (*lld_v*,*) containing the local pieces of the global distributed matrix *V* representing the Householder transformation *Q*,

V(*iv*:*iv*+l-1, *jv*) if *side* = 'L' and *incv* = 1,

V(*iv*, *jv*:*jv*+*l*-1) if *side* = 'L' and *incv* = *m_v*,

V(*iv*:*iv*+*l*-1, *jv*) if *side* = 'R' and *incv* = 1,

V(*iv*, *jv*:*jv*+*l*-1) if *side* = 'R' and *incv* = *m_v*.

The vector *v* in the representation of *Q*. *v* is not used if *tau* = 0.

iv, *jv* (global) INTEGER.

The row and column indices in the global matrix *V* indicating the first row and the first column of the matrix sub(*V*), respectively.

descv (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *V*.

incv (global) INTEGER.

The global increment for the elements of *V*. Only two values of *incv* are supported in this version, namely 1 and *m_v*.

incv must not be zero.

tau (local)

COMPLEX for pclarzc

COMPLEX*16 for pzlarzc.

Array of size *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

c (local).

COMPLEX for pclarzc

COMPLEX*16 for pzlarzc.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic, *jc* (global) INTEGER.

The row and column indices in the global matrix *C* indicating the first row and the first column of the matrix sub(*C*), respectively.

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local).

```
If incv = 1,
  if side = 'L' ,
    if ivcol = iccol,
      lwork≥ nqc0
    else
      lwork ≥ mpc0 + max(1, nqc0)
    end if
  else if side = 'R' ,
    lwork ≥nqc0 + max(max(1, mpc0), numroc(numroc(n+icoffc, nb_v,
0, 0, npcol),
           nb_v, 0, 0, lcmq))  end if
else if incv = m_v,
  if side = 'L' ,
    lwork≥mpc0 + max(max(1, nqc0), numroc(numroc(m+iroffc, mb_v,
0, 0, nprow),
           mb_v, 0, 0, lcmp))
  else if side = 'R',
    if ivrow = icrow,
      lwork≥ mpc0
    else
      lwork≥nqc0 + max(1, mpc0)
    end if
  end if
        end if
```

Here *lcm* is the least common multiple of *nprow* and *npcol*;

lcm = ilcm(*nprow*, *npcol*), *lcmp* = *lcm*/*nprow*, *lcmq*= *lcm*/*npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc*= mod(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

mpc0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

nqc0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions;

myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).

On exit, sub(*C*) is overwritten by the Q^H*sub(*C*) if *side* = 'L', or sub(*C*)*Q^H if *side* = 'R'.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larzt

*Forms the triangular factor T of a block reflector H=I-V*T*V^H as returned by* p?tzrzf.

Syntax

```
call pslarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pdlarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pclarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pzlarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

Description

The p?larztroutine forms the triangular factor *T* of a real/complex block reflector *H* of order greater than *n*, which is defined as a product of *k* elementary reflectors as returned by p?tzrzf.

If *direct* = 'F', *H* = *H*(1)**H*(2)*...**H*(*k*), and *T* is upper triangular;

If *direct* = 'B', *H* = *H*(*k*)*...**H*(2)**H*(1), and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector *H*(*i*), is stored in the *i*-th column of the array *v*, and

H = *i*-*v***t***v*'.

If *storev* = 'R', the vector, which defines the elementary reflector *H*(*i*), is stored in the *i*-th row of the array *v*, and

H = *i*-*v*'**t***v*

Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

direct

(global) CHARACTER.

Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

if *direct* = 'F': *H* = *H*(1)**H*(2)*...**H*(*k*) (Forward, not supported)

if *direct* = 'B': *H* = *H*(*k*)*...**H*(2)**H*(1) (Backward).

storev

(global) CHARACTER.

Specifies how the vectors which defines the elementary reflectors are stored:

if *storev* = 'C': columnwise (not supported);

if *storev* = 'R': rowwise.

n

(global) INTEGER.

The order of the block reflector *H*. *n* ≥ 0.

k

(global) INTEGER.

The order of the triangular factor *T* (= the number of elementary reflectors).

1≤*k*≤*mb_v*(= *nb_v*).

v

REAL for pslarzt

DOUBLE PRECISION for pdlarzt

COMPLEX for pclarzt

COMPLEX*16 for pzlarzt.

Pointer into the local memory to an array of local size (*LOCr*(*iv*+*k*-1), *LOCc*(*jv*+*n*-1)).

The distributed matrix *V* contains the Householder vectors. See *Application Notes* below.

iv, *jv*

(global) INTEGER. The row and column indices in the global matrix *V* indicating the first row and the first column of the matrix sub(*V*), respectively.

descv

(local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *V*.

tau

(local)

REAL for pslarzt

DOUBLE PRECISION for pdlarzt

COMPLEX for pclarzt

COMPLEX*16 for pzlarzt.

Array of size *LOCr*(*iv*+*k*-1) if *incv* = *m_v*, and *LOCc*(*jv*+*k*-1) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

work

(local).

REAL for pslarzt

DOUBLE PRECISION for pdlarzt

COMPLEX for pclarzt

COMPLEX*16 for pzlarzt.

Workspace array of size (*k**(*k*-1)/2).

Output Parameters

v

REAL for pslarzt

DOUBLE PRECISION for pdlarzt

COMPLEX for pclarzt

COMPLEX*16 for pzlarzt.

t

(local)

REAL for pslarzt

DOUBLE PRECISION for pdlarzt

COMPLEX for pclarzt

COMPLEX*16 for pzlarzt.

Array of size *mb_v*by *mb_v*. It contains the *k*-by-*k* triangular factor of the block reflector associated with *v*. *t* is lower triangular.

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the `H(i)` is best illustrated by the following example with `n = 5` and `k = 3`. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct='F' and *storev*='C'

$$v = \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix} \\ \begin{matrix} . & . & . \\ . & . & . \\ 1 & . & . \\ & 1 & . \\ & & 1 \end{matrix}$$

direct='F' and *storev*='R':

$$\begin{bmatrix} \overbrace{\begin{matrix} v1 & v1 & v1 & v1 & v1 \\ v2 & v2 & v2 & v2 & v2 \\ v3 & v3 & v3 & v3 & v3 \end{matrix}}^{V} & \begin{matrix} . & . & . & . & 1 \\ . & . & . & 1 & \\ . & . & 1 & & \end{matrix} \end{bmatrix}$$

direct='B' and storev='C':

$$V = \begin{bmatrix} 1 & & \\ . & 1 & \\ . & . & 1 \\ . & . & . \\ . & . & . \end{bmatrix} \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}$$

direct='B' and storev='R':

$$\begin{bmatrix} & & & & \overbrace{\qquad\qquad\qquad\qquad}^{V} & & & & \\ 1 & . & . & . & v1 & v1 & v1 & v1 & v1 \\ . & 1 & . & . & v2 & v2 & v2 & v2 & v2 \\ . & . & 1 & . & v3 & v3 & v3 & v3 & v3 \end{bmatrix}$$

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lascl

Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from}.

Syntax

```
call pslascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pdlascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pclascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pzlascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
```

Description

The `p?lascl`routine multiplies the *m*-by-*n* real/complex distributed matrix sub(*A*) denoting *A*`(ia:ia+m-1, ja:ja+n-1)` by the real/complex scalar `cto/cfrom`. This is done without over/underflow as long as the final result `cto`**A*(*i*,*j*)/`cfrom` does not over/underflow. `type` specifies that sub(*A*) may be full, upper triangular, lower triangular or upper Hessenberg.

Input Parameters

type (global) CHARACTER.

type indicates the storage type of the input distributed matrix.

if *type* = 'G': sub(*A*) is a full matrix,

if *type* = 'L': sub(*A*) is a lower triangular matrix,

if *type* = 'U': sub(*A*) is an upper triangular matrix,

if *type* = 'H': sub(*A*) is an upper Hessenberg matrix.

cfrom, *cto* (global)

REAL for pslascl/pclascl

DOUBLE PRECISION for pdlascl/pzlascl.

The distributed matrix sub(*A*) is multiplied by *cto/cfrom*. *A*(*i*,*j*) is computed without over/underflow if the final result *cto***A*(*i*,*j*)/*cfrom* can be represented without over/underflow. *cfrom* must be nonzero.

m (global) INTEGER.

The number of rows in the distributed matrix sub(*A*). (*m*≥0).

n (global) INTEGER.

The number of columns in the distributed matrix sub(*A*). (*n*≥0).

a (local input/local output)

REAL for pslascl

DOUBLE PRECISION for pdlascl

COMPLEX for pclascl

COMPLEX*16 for pzlascl.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

This array contains the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER.

The column and row indices in the global matrix *A* indicating the first row and column of the matrix sub(*A*), respectively.

desca (global and local) INTEGER.

Array of size *dlen_*.The array descriptor for the distributed matrix *A*.

Output Parameters

a (local).

On exit, this array contains the local pieces of the distributed matrix multiplied by *cto/cfrom*.

info (local)

INTEGER.

if *info* = 0: the execution is successful.

if *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*),

if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lase2

Initializes an m-by-n distributed matrix.

Syntax

```
call pslase2 (uplo, m, n, alpha, beta, a, ia, ja, desca )
call pdlase2 (uplo, m, n, alpha, beta, a, ia, ja, desca )
call pclase2 (uplo, m, n, alpha, beta, a, ia, ja, desca )
call pzlase2 (uplo, m, n, alpha, beta, a, ia, ja, desca )
```

Description

p?lase2 initializes an *m*-by-*n* distributed matrix sub(*A*) denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) to *beta* on the diagonal and *alpha* on the off-diagonals. p?lase2 requires that only the dimension of the matrix operand is distributed.

Input Parameters

uplo (global)

CHARACTER.

Specifies the part of the distributed matrix sub(*A*) to be set:

= 'U': Upper triangular part is set; the strictly lower triangular part of sub(*A*) is not changed;

= 'L': Lower triangular part is set; the strictly upper triangular part of sub(*A*) is not changed;

Otherwise: All of the matrix sub(*A*) is set.

m (global)

INTEGER.

The number of rows to be operated on i.e the number of rows of the distributed submatrix sub(*A*). *m* >= 0.

n (global)

INTEGER.

The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(*A*). *n* >= 0.

alpha (global)

REAL for pslase2

DOUBLE PRECISION for pdlase2

COMPLEX for pclase2

DOUBLE COMPLEX for pzlase2

The constant to which the off-diagonal elements are to be set.

beta (global)

REAL for pslase2

DOUBLE PRECISION for pdlase2

COMPLEX for pclase2

DOUBLE COMPLEX for pzlase2

The constant to which the diagonal elements are to be set.

ia (global)

INTEGER.

The row index in the global array *a* indicating the first row of sub(*A*).

ja (global)

INTEGER.

The column index in the global array *a* indicating the first column of sub(*A*).

desca (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

Output Parameters

a (local)

REAL for pslase2

DOUBLE PRECISION for pdlase2

COMPLEX for pclase2

DOUBLE COMPLEX for pzlase2

Pointer into the local memory to an array of size (*lld_a*,*LOCc*(*ja*+*n*-1)).

This array contains the local pieces of the distributed matrix sub(*A*) to be set.

On exit, the leading *m*-by-*n* submatrix sub(*A*) is set as follows:

if *uplo* = 'U', *A*(*ia*+i-1,*ja*+j-1) = *alpha*, 1<=i<=j-1, 1<=j<=*n*,

if *uplo* = 'L', *A*(*ia*+i-1,*ja*+j-1) = *alpha*, j+1<=i<=*m*, 1<=j<=*n*,

otherwise, *A*(*ia*+i-1,*ja*+j-1) = *alpha*, 1<=i<=*m*, 1<=j<=*n*, *ia*+i != *ja*+j,

and, for all *uplo*, *A*(*ia*+i-1,*ja*+i-1) = *beta*, 1<=i<=min(*m*,*n*).

p?laset

Initializes the offdiagonal elements of a matrix to alpha and the diagonal elements to beta.

Syntax

```
call pslaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pdlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pclaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pzlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
```

Description

The p?lasetroutine initializes an *m*-by-*n* distributed matrix sub(*A*) denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) to *beta* on the diagonal and *alpha* on the offdiagonals.

Input Parameters

uplo

(global) CHARACTER.

Specifies the part of the distributed matrix sub(*A*) to be set:

if *uplo* = 'U': upper triangular part; the strictly lower triangular part of sub(*A*) is not changed;

if *uplo* = 'L': lower triangular part; the strictly upper triangular part of sub(*A*) is not changed.

Otherwise: all of the matrix sub(*A*) is set.

m

(global) INTEGER.

The number of rows in the distributed matrix sub(*A*). (*m*≥0).

n

(global) INTEGER.

The number of columns in the distributed matrix sub(*A*). (*n*≥0).

alpha

(global).

REAL for pslaset

DOUBLE PRECISION for pdlaset

COMPLEX for pclaset

COMPLEX*16 for pzlaset.

The constant to which the offdiagonal elements are to be set.

beta

(global).

REAL for pslaset

DOUBLE PRECISION for pdlaset

COMPLEX for pclaset

COMPLEX*16 for pzlaset.

The constant to which the diagonal elements are to be set.

Output Parameters

a (local).

`REAL` for `pslaset`

`DOUBLE PRECISION` for `pdlaset`

`COMPLEX` for `pclaset`

`COMPLEX*16` for `pzlaset`.

Pointer into the local memory to an array of size `(lld_a, LOCc(ja+n-1))`.

This array contains the local pieces of the distributed matrix sub(*A*) to be set. On exit, the leading *m*-by-*n* matrix sub(*A*) is set as follows:

if `uplo = 'U'`, *A*(`ia`+*i*-1, `ja`+*j*-1) = `alpha`, 1≤*i*≤*j*-1, 1≤*j*≤`n`,

if `uplo = 'L'`, *A*(`ia`+*i*-1, `ja`+*j*-1) = `alpha`, *j*+1≤*i*≤ `m`, 1≤*j*≤`n`,

otherwise, *A*(`ia`+*i*-1, `ja`+*j*-1) = `alpha`, 1≤*i*≤`m`, 1≤*j*≤`n`, `ia`+*i*≠`ja`+*j*, and, for all `uplo`, *A*(`ia`+*i*-1, `ja`+*i*-1) = `beta`, 1≤*i*≤min(`m`,`n`).

ia, *ja* (global) `INTEGER`.

The column and row indices in the distributed matrix *A* indicating the first row and column of the matrix sub(*A*), respectively.

desca (global and local)`INTEGER`.

Array of size *dlen_*. The array descriptor for the distributed matrix *A*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lasmsub

Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.

Syntax

```
call pslasmsub(a, desca, i, l, k, smlnum, buf, lwork)
call pdlasmsub(a, desca, i, l, k, smlnum, buf, lwork)
call pclasmsub(a, desca, i, l, k, smlnum, buf, lwork)
call pzlasmsub(a, desca, i, l, k, smlnum, buf, lwork)
```

Description

The `p?lasmsub`routine looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero. This routine performs a global maximum and must be called by all processes.

Input Parameters

a (local)

`REAL` for `pslasmsub`

`DOUBLE PRECISION` for `pdlasmsub`

`COMPLEX` for `pclasmsub`

DOUBLE COMPLEX for pzlasmsub

Array of size (*lld_a*,*LOCc*(*n_a*)).

On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.

desca (global and local) INTEGER.

Array of size *dlen_*. The array descriptor for the distributed matrix *A*.

i (global) INTEGER.

The global location of the bottom of the unreduced submatrix of *A*. Unchanged on exit.

l (global) INTEGER.

The global location of the top of the unreduced submatrix of *A*.

Unchanged on exit.

smlnum (global)

REAL for pslasmsub

DOUBLE PRECISION for pdlasmsub

REAL for pclasmsub

DOUBLE PRECISION for pzlasmsub

On entry, a "small number" for the given matrix. Unchanged on exit. The machine-dependent constants for the stopping criterion.

lwork (local) INTEGER.

This must be at least 2*ceil(ceil((*i*-1)/*mb_a*)/ lcm(*nprow*,*npcol*)). Here lcm is least common multiple and *nprowxnpcol* is the logical grid size.

Output Parameters

k (global) INTEGER.

On exit, this yields the bottom portion of the unreduced submatrix. This will satisfy: $l \leq k \leq i-1$.

buf (local).

REAL for pslasmsub

DOUBLE PRECISION for pdlasmsub

COMPLEX for pclasmsub

DOUBLE COMPLEX for pzlasmsub

Array of size *lwork*.

Application Notes

This routine parallelizes the code from ?lahqr that looks for a single small subdiagonal element.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lasrt

Sorts the numbers in an array and the corresponding vectors in increasing order.

Syntax

```
call pslasrt (id, n, d, q, iq, jq, descq, work, lwork, iwork, liwork, info )
call pdlasrt (id, n, d, q, iq, jq, descq, work, lwork, iwork, liwork, info )
```

Description

p?lasrt sorts the numbers in *d* and the corresponding vectors in *q* in increasing order.

Input Parameters

id (global)

CHARACTER*1.

= 'I': sort *d* in increasing order;

= 'D': sort *d* in decreasing order. (NOT IMPLEMENTED YET)

n (global)

INTEGER.

The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(*Q*). *n* >= 0.

d (global)

REAL for pslasrt

DOUBLE PRECISION for pdlasrt

Array, size (*n*)

q (local)

REAL for pslasrt

DOUBLE PRECISION for pdlasrt

Pointer into the local memory to an array of size (*lld_q*,*LOCc*(*jq*+*n*-1)) .

This array contains the local pieces of the distributed matrix sub(*A*) to be copied from.

iq (global)

INTEGER.

The row index in the global array A indicating the first row of sub(*Q*).

jq (global)

INTEGER.

The column index in the global array A indicating the first column of sub(*Q*).

descq (global and local)

INTEGER.

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

work (local)

`REAL` for `pslasrt`

`DOUBLE PRECISION` for `pdlasrt`

Array, size (*lwork*)

lwork (local)

INTEGER.

The size of the array *work*.

lwork = MAX(*n*, NP * (NB + NQ)), where NP = `numroc`(*n*, NB, MYROW, IAROW, NPROW), NQ = `numroc`(*n*, NB, MYCOL, DESCQ(*csrc_*), NPCOL).

`numroc` is a ScaLAPACK tool function.

iwork (local)

INTEGER.

Array, size (*liwork*)

liwork (local)

INTEGER.

The size of the array *iwork*.

liwork = *n* + 2*NB + 2*NPCOL

Output Parameters

d On exit, the numbers in *d* are sorted in increasing order.

info (global)

INTEGER.

= 0: successful exit

< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call pslassq(n, x, ix, jx, descx, incx, scale, sumsq)
call pdlassq(n, x, ix, jx, descx, incx, scale, sumsq)
call pclassq(n, x, ix, jx, descx, incx, scale, sumsq)
call pzlassq(n, x, ix, jx, descx, incx, scale, sumsq)
```

Description

The p?lassqroutine returns the values *scl* and *smsq* such that

$scl^2 * smsq = x_1^2 + ... + x_n^2 + scale^2 * sumsq$,

where

x_i= sub(X) = X(*ix* + (*jx*-1)**m_x* + (*i* - 1)**incx*) for pslassq/pdlassq ,

x_i= sub(X) = abs(X(*ix* + (*jx*-1)**m_x* + (*i* - 1)**incx*) for pclassq/pzlassq.

For real routines pslassq/pdlassq the value of *sumsq* is assumed to be non-negative and *scl* returns the value

scl = max(*scale*, abs(x_i)).

For complex routines pclassq/pzlassq the value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy

$1.0 \le ssq \le sumsq + 2n$

Value *scale* is assumed to be non-negative and *scl* returns the value

$$scl = \max_i \left(scale, \mathbf{abs}\left(\mathbf{real}\left(x_i\right)\right), \mathbf{abs}\left(\mathbf{aimag}\left(x_i\right)\right)\right)$$

For all routines p?lassq values *scale* and *sumsq* must be supplied in *scale* and *sumsq* respectively, and *scale* and *sumsq* are overwritten by *scl* and *ssq* respectively.

All routines p?lassq make only one pass through the vector sub(X).

Input Parameters

n
(global) INTEGER.
The length of the distributed vector sub(x).

x
REAL for pslassq
DOUBLE PRECISION for pdlassq
COMPLEX for pclassq
COMPLEX*16 for pzlassq.
The vector for which a scaled sum of squares is computed:
x(*ix* + (*jx*-1)**m_x* + (*i* - 1)**incx*), 1 ≤ *i* ≤ *n*.

ix
(global) INTEGER.
The row index in the global matrix X indicating the first row of sub(X).

jx
(global) INTEGER.
The column index in the global matrix X indicating the first column of sub(X).

descx
(global and local) INTEGER array of size *dlen_*.
The array descriptor for the distributed matrix X.

incx
(global) INTEGER.

	The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. The argument *incx* must not equal zero.
scale	(local). `REAL` for `pslassq/pclassq` `DOUBLE PRECISION` for `pdlassq/pzlassq`. On entry, the value *scale* in the equation above.
sumsq	(local) `REAL` for `pslassq/pclassq` `DOUBLE PRECISION` for `pdlassq/pzlassq`. On entry, the value *sumsq* in the equation above.

Output Parameters

scale	(local). On exit, *scale* is overwritten with *scl* , the scaling factor for the sum of squares.
sumsq	(local). On exit, *sumsq* is overwritten with the value *smsq,* the basic sum of squares from which *scl* has been factored out.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call pslaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pdlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pclaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pzlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
```

Description

The `p?laswp`routine performs a series of row or column interchanges on the distributed matrix `sub(A)=A(ia:ia+n-1, ja:ja+n-1)`. One interchange is initiated for each of rows or columns *k1* through *k2* of sub(*A*). This routine assumes that the pivoting information has already been broadcast along the process row or column. Also note that this routine will only work for *k1-k2* being in the same *mb* (or *nb*) block. If you want to pivot a full matrix, use `p?lapiv`.

Input Parameters

direc	(global) `CHARACTER`. Specifies in which order the permutation is applied:

= 'F' - forward,

= 'B' - backward.

rowcol (global) CHARACTER.

Specifies if the rows or columns are permuted:

= 'R' - rows,

= 'C' - columns.

n (global) INTEGER.

If *rowcol*='R', the length of the rows of the distributed matrix *A*(*, *ja*:*ja*+*n*-1) to be permuted;

If *rowcol*='C', the length of the columns of the distributed matrix *A*(*ia*:*ia*+*n*-1 , *) to be permuted;

a (local)

REAL for pslaswp

DOUBLE PRECISION for pdlaswp

COMPLEX for pclaswp

COMPLEX*16 for pzlaswp.

Pointer into the local memory to an array of size (*lld_a*, *). On entry, this array contains the local pieces of the distributed matrix to which the row/columns interchanges will be applied.

ia (global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja (global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

k1 (global) INTEGER.

The first element of *ipiv* for which a row or column interchange will be done.

k2 (global) INTEGER.

The last element of *ipiv* for which a row or column interchange will be done.

ipiv (local)

INTEGER. Array of size *LOCr*(*m_a*)+*mb_a* for row pivoting and *LOCr*(*n_a*)+*nb_a* for column pivoting. This array is tied to the matrix *A*, *ipiv*(*k*)=*l* implies rows (or columns) *k* and *l* are to be interchanged.

Output Parameters

A (local)

REAL for pslaswp

DOUBLE PRECISION for pdlaswp

COMPLEX for pclaswp

COMPLEX*16 for pzlaswp.

On exit, the permuted distributed matrix.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?latra

Computes the trace of a general square distributed matrix.

Syntax

```
val = pslatra(n, a, ia, ja, desca)
val = pdlatra(n, a, ia, ja, desca)
val = pclatra(n, a, ia, ja, desca)
val = pzlatra(n, a, ia, ja, desca)
```

Description

This function computes the trace of an *n*-by-*n* distributed matrix sub(*A*) denoting A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1). The result is left on every process of the grid.

Input Parameters

n — (global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(*A*). *n* ≥0.

a — (local).

REAL for pslatra

DOUBLE PRECISION for pdlatra

COMPLEX for pclatra

COMPLEX*16 for pzlatra.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix, the trace of which is to be computed.

ia, *ja* — (global) INTEGER. The row and column indices respectively in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca — (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

val — The value returned by the function.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
call pslatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pdlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pclatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pzlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
```

Description

The p?latrdroutine reduces *nb* rows and columns of a real symmetric or complex Hermitian matrix sub(*A*)= *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) to symmetric/complex tridiagonal form by an orthogonal/unitary similarity transformation *Q*'*sub(*A*)**Q*, and returns the matrices *V* and *W*, which are needed to apply the transformation to the unreduced part of sub(*A*).

If *uplo* = U, p?latrd reduces the last *nb* rows and columns of a matrix, of which the upper triangle is supplied;

if *uplo* = L, p?latrd reduces the first *nb* rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by p?sytrd/p?hetrd.

Input Parameters

uplo

(global) CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix sub(*A*) is stored:

= 'U': Upper triangular

= L: Lower triangular.

n

(global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(*A*). *n* ≥ 0.

nb

(global) INTEGER.

The number of rows and columns to be reduced.

a

REAL for pslatrd

DOUBLE PRECISION for pdlatrd

COMPLEX for pclatrd

COMPLEX*16 for pzlatrd.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the symmetric/Hermitian distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia (global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja (global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

iw (global) INTEGER.

The row index in the global matrix *W* indicating the first row of sub(*W*).

jw (global) INTEGER.

The column index in the global matrix *W* indicating the first column of sub(*W*).

descw (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *W*.

work (local)

REAL for pslatrd

DOUBLE PRECISION for pdlatrd

COMPLEX for pclatrd

COMPLEX*16 for pzlatrd.

Workspace array of size *nb_a*.

Output Parameters

a (local)

On exit, if *uplo* = 'U', the last *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of sub(*A*); the elements above the diagonal with the array *tau* represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;

if *uplo* = 'L', the first *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of sub(*A*); the elements below the diagonal with the array *tau* represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

d (local)

REAL for pslatrd/pclatrd

DOUBLE PRECISION for pdlatrd/pzlatrd.

Array of size *LOCc(ja+n-1)*.

The diagonal elements of the tridiagonal matrix *T*: *d*(i) = *a*(i,i). *d* is tied to the distributed matrix *A*.

e

(local)

REAL for pslatrd/pclatrd

DOUBLE PRECISION for pdlatrd/pzlatrd.

Array of size *LOCc(ja+n-1)* if *uplo* = 'U', *LOCc(ja+n-2)* otherwise.

The off-diagonal elements of the tridiagonal matrix *T*:

e(*i*) = *a*(*i*, *i* + 1) if *uplo* = 'U',

e(*i*) = *a*(*i* + 1, *i*) if *uplo* = 'L'.

e is tied to the distributed matrix *A*.

tau

(local)

REAL for pslatrd

DOUBLE PRECISION for pdlatrd

COMPLEX for pclatrd

COMPLEX*16 for pzlatrd.

Array of size *LOCc(ja+n-1)*. This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

w

(local)

REAL for pslatrd

DOUBLE PRECISION for pdlatrd

COMPLEX for pclatrd

COMPLEX*16 for pzlatrd.

Pointer into the local memory to an array of size *lld_w*by *nb_w*. This array contains the local pieces of the *n*-by-*nb_w* matrix *w* required to update the unreduced part of sub(*A*).

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

```
Q = H(n)*H(n-1)*...*H(n-nb+1)
```

Each *H*(*i*) has the form

```
H(i) = I - tau*v*v' ,
```

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*i*:*n*) = 0 and *v*(*i*-1) = 1; *v*(1:*i*-1) is stored on exit in *A*(*ia*:*ia*+*i*-1, *ja*+*i*), and *tau* in *tau*(*ja*+*i*-1).

If *uplo* = L, the matrix *Q* is represented as a product of elementary reflectors

```
Q = H(1)*H(2)*...*H(nb)
```

Each *H*(i) has the form

```
H(i) = I - tau*v*v' ,
```

where *tau* is a real/complex scalar, and *v* is a real/complex vector with `v(1:i) = 0` and `v(i+1) = 1`; `v(i+2: n)` is stored on exit in `A(ia+i+1: ia+n-1, ja+i-1)`, and *tau* in `tau(ja+i-1)`.

The elements of the vectors *v* together form the *n*-by-*nb* matrix *V* which is needed, with *W*, to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2*k* update of the form:

`sub(A) := sub(A)-vw'-wv'`.

The contents of *a* on exit are illustrated by the following examples with

`n = 5` and `nb = 2`:

if *uplo*='U':

$$\begin{bmatrix} a & a & a & v_4 & v_5 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & & d & 1 \\ & & & & d \end{bmatrix}$$

if *uplo*='L':

$$\begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where *d* denotes a diagonal element of the reduced matrix, *a* denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining *H*(i).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

`call pslatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale, cnorm, work)`

`call pdlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale, cnorm, work)`

`call pclatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale, cnorm, work)`

`call pzlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale, cnorm, work)`

Description

The `p?latrs`routine solves a triangular system of equations $Ax = sb$, $A^Tx = sb$ or $A^Hx = sb$, where `s` is a scale factor set to prevent overflow. The description of the routine will be extended in the future releases.

Input Parameters

uplo CHARACTER*1.

Specifies whether the matrix *A* is upper or lower triangular.

= 'U': Upper triangular

= 'L': Lower triangular

trans

CHARACTER*1.

Specifies the operation applied to *Ax*.

= 'N': Solve $Ax = s*b$ (no transpose)

= 'T': Solve $A^Tx = s*b$ (transpose)

= 'C': Solve $A^Hx = s*b$ (conjugate transpose),

where *s* - is a scale factor

diag

CHARACTER*1.

Specifies whether or not the matrix *A* is unit triangular.

= 'N': Non-unit triangular

= 'U': Unit triangular

normin

CHARACTER*1.

Specifies whether *cnorm* has been set or not.

= 'Y': *cnorm* contains the column norms on entry;

= 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in *cnorm*.

n

INTEGER.

The order of the matrix *A*. $n \geq 0$

a

REAL for pslatrs/pclatrs

DOUBLE PRECISION for pdlatrs/pzlatrs

Array of size *lda*by *n*. Contains the triangular matrix *A*.

If *uplo* = U, the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced.

If *diag* = 'U', the diagonal elements of *a* are also not referenced and are assumed to be 1.

ia, *ja*

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

x

REAL for pslatrs/pclatrs

DOUBLE PRECISION for pdlatrs/pzlatrs

	Array of size *n*. On entry, the right hand side *b* of the triangular system.
ix	(global)INTEGER.The row index in the global matrix *X* indicating the first row of sub(*x*).
jx	(global) INTEGER. The column index in the global matrix *X* indicating the first column of sub(*X*).
descx	(global and local) INTEGER. Array of size *dlen_*. The array descriptor for the distributed matrix *X*.
cnorm	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs. Array of size *n*. If *normin* = 'Y', *cnorm* is an input argument and *cnorm*(*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*. If *trans* = 'N', *cnorm*(*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.
work	(local). REAL for pslatrs DOUBLE PRECISION for pdlatrs COMPLEX for pclatrs COMPLEX*16 for pzlatrs. Temporary workspace.

Output Parameters

X	On exit, *x* is overwritten by the solution vector *x*.
scale	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs. Array of size *lda*by *n*. The scaling factor *s* for the triangular system as described above. If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to *Ax* = 0.
cnorm	If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?latrz

Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.

Syntax

```
call pslatrz(m, n, l, a, ia, ja, desca, tau, work)
```

```
call pdlatrz(m, n, l, a, ia, ja, desca, tau, work)
call pclatrz(m, n, l, a, ia, ja, desca, tau, work)
call pzlatrz(m, n, l, a, ia, ja, desca, tau, work)
```

Description

The p?latrzroutine reduces the *m*-by-*n* (*m* ≤ *n*) real/complex upper trapezoidal matrix sub(*A*) = [*A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*m*-1)*A*(*ia*:*ia*+*m*-1, *ja*+*n*-*l*:*ja*+*n*-1)] to upper triangular form by means of orthogonal/unitary transformations.

The upper trapezoidal matrix sub(*A*) is factored as

sub(*A*) = (*R* 0)**Z*,

where *Z* is an *n*-by-*n* orthogonal/unitary matrix and *R* is an *m*-by-*m* upper triangular matrix.

Input Parameters

m

(global) INTEGER.

The number of rows in the distributed matrix sub(*A*). *m* ≥ 0.

n

(global) INTEGER.

The number of columns in the distributed matrix sub(*A*). *n* ≥ 0.

l

(global) INTEGER.

The number of columns of the distributed matrix sub(*A*) containing the meaningful part of the Householder reflectors. *l* > 0.

a

(local)

REAL for pslatrz

DOUBLE PRECISION for pdlatrz

COMPLEX for pclatrz

COMPLEX*16 for pzlatrz.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the local pieces of the *m*-by-*n* distributed matrix sub(*A*), which is to be factored.

ia

(global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja

(global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

work

(local)

REAL for pslatrz

DOUBLE PRECISION for pdlatrz

COMPLEX for pclatrz

COMPLEX*16 for pzlatrz.

Workspace array of size *lwork*.

lwork ≥ *nq*0 + max(1, *mp*0), where

iroff = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),

numroc, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

a

On exit, the leading *m*-by-*m* upper triangular part of sub(*A*) contains the upper triangular matrix *R*, and elements *n*-*l*+1 to *n* of the first *m* rows of sub(*A*), with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

tau

(local)

REAL for pslatrz

DOUBLE PRECISION for pdlatrz

COMPLEX for pclatrz

COMPLEX*16 for pzlatrz.

Array of size*LOCr*(*ja*+*m*-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is used (or, in case of complex routines, whose conjugate transpose is used) to introduce zeros into the (*m* - *k* + 1)-th row of sub(*A*), is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix},$$

where

$$T(k) = I - tau * u(k) * u(k)', \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and `z( k )` is an (*n*-*m*)-element vector. *tau* and `z( k )` are chosen to annihilate the elements of the *k*-th row of sub(*A*). The scalar *tau* is returned in the *k*-th element of *tau* and the vector `u( k )` in the *k*-th row of sub(*A*), such that the elements of `z(k )` are in *A*`( k, m + 1 ), ..., A( k, n )`. The elements of *R* are returned in the upper triangular part of sub(*A*).

Z is given by

```
Z =  Z(1)Z(2)...Z(m).
```

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lauu2

Computes the product `U*U'` or `L'*L`, where U and L are upper or lower triangular matrices (local unblocked algorithm).

Syntax

```
call pslauu2(uplo, n, a, ia, ja, desca)
call pdlauu2(uplo, n, a, ia, ja, desca)
call pclauu2(uplo, n, a, ia, ja, desca)
call pzlauu2(uplo, n, a, ia, ja, desca)
```

Description

The `p?lauu2`routine computes the product *U***U*' or *L*'**L*, where the triangular factor *U* or *L* is stored in the upper or lower triangular part of the distributed matrix

`sub(A)= A(ia:ia+n-1, ja:ja+n-1).`

If *uplo* = `'U'` or `'u'`, then the upper triangle of the result is stored, overwriting the factor *U* in sub(*A*).

If *uplo* = `'L'` or `'l'`, then the lower triangle of the result is stored, overwriting the factor *L* in sub(*A*).

This is the unblocked form of the algorithm, calling BLAS Level 2 Routines. No communication is performed by this routine, the matrix to operate on should be strictly local to one process.

Input Parameters

uplo

(global) `CHARACTER*1`.

Specifies whether the triangular factor stored in the *matrix* sub(*A*) is upper or lower triangular:

= `U`: upper triangular

= `L`: lower triangular.

n

(global) `INTEGER`.

The number of rows and columns to be operated on, that is, the order of the triangular factor *U* or *L*. $n \geq 0$.

a

(local)

`REAL` for `pslauu2`

`DOUBLE PRECISION` for `pdlauu2`

`COMPLEX` for `pclauu2`

COMPLEX*16 for pzlauu2.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the local pieces of the triangular factor *U* or *L*.

ia (global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja (global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a (local)

On exit, if *uplo* = 'U', the upper triangle of the distributed matrix sub(*A*) is overwritten with the upper triangle of the product *U***U*'; if *uplo* = 'L', the lower triangle of sub(*A*) is overwritten with the lower triangle of the product *L*'**L*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lauum

*Computes the product U*U' or L'*L, where U and L are upper or lower triangular matrices.*

Syntax

```
call pslauum(uplo, n, a, ia, ja, desca)
call pdlauum(uplo, n, a, ia, ja, desca)
call pclauum(uplo, n, a, ia, ja, desca)
call pzlauum(uplo, n, a, ia, ja, desca)
```

Description

The p?lauumroutine computes the product *U***U*' or *L*'**L*, where the triangular factor *U* or *L* is stored in the upper or lower triangular part of the matrix sub(*A*)= *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in sub(*A*). If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in sub(*A*).

This is the blocked form of the algorithm, calling Level 3 PBLAS.

Input Parameters

uplo (global) CHARACTER*1.

Specifies whether the triangular factor stored in the matrix sub(*A*) is upper or lower triangular:

= 'U': upper triangular

= 'L': lower triangular.

n (global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the triangular factor *U* or *L*. *n* ≥ 0.

a (local)

REAL for pslauum

DOUBLE PRECISION for pdlauum

COMPLEX for pclauum

COMPLEX*16 for pzlauum.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the local pieces of the triangular factor *U* or *L*.

ia (global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja (global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a (local)

On exit, if *uplo* = 'U', the upper triangle of the distributed matrix sub(*A*) is overwritten with the upper triangle of the product *U***U*' ; if *uplo* = 'L', the lower triangle of sub(*A*) is overwritten with the lower triangle of the product *L*'**L*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lawil

Forms the Wilkinson transform.

Syntax

```
call pslawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
call pdlawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
call pclawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
call pzlawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
```

Description

The p?lawilroutine gets the transform given by *h44*, *h33*, and *h43h34* into *v* starting at row *m*.

Input Parameters

ii

(global) INTEGER.

Number of the process row which owns the matrix element *A*(*m*+2, *m*+2).

jj

(global) INTEGER.

Number of the process column which owns the matrix element *A*(*m*+2, *m* +2).

m

(global) INTEGER.

On entry, the location from where the transform starts (row *m*). Unchanged on exit.

a

(local)

REAL for pslawil

DOUBLE PRECISION for pdlawil

COMPLEX for pclawil

DOUBLE COMPLEX for pzlawil

Array of size (*lld_a*,*LOCc*(*n_a*)).

On entry, the Hessenberg matrix. Unchanged on exit.

desca

(global and local) INTEGER

Array of size *dlen_*. The array descriptor for the distributed matrix *A*. Unchanged on exit.

h43h34

(global)

REAL for pslawil

DOUBLE PRECISION for pdlawil

COMPLEX for pclawil

DOUBLE COMPLEX for pzlawil

These three values are for the double shift *QR* iteration. Unchanged on exit.

Output Parameters

v

(global)

REAL for pslawil

DOUBLE PRECISION for pdlawil

COMPLEX for pclawil

DOUBLE COMPLEX for pzlawil

Array of size 3 that contains the transform on output.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?org2l/p?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by `p?geqlf` *(unblocked algorithm).*

Syntax

```
call psorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The `p?org2l/p?ung2l`routine generates an *m*-by-*n* real/complex distributed matrix *Q* denoting *A*`(ia:ia+m-1, ja:ja+n-1)` with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors of order *m*:

Q = *H*(*k*)*...*H*(2)*H*(1) as returned by `p?geqlf`.

Input Parameters

m

(global) `INTEGER`.

The number of rows in the distributed submatrix *Q*. $m \geq 0$.

n

(global) `INTEGER`.

The number of columns in the distributed submatrix *Q*. $m \geq n \geq 0$.

k

(global) `INTEGER`.

The number of elementary reflectors whose product defines the matrix *Q*. $n \geq k \geq 0$.

a

`REAL` for `psorg2l`

`DOUBLE PRECISION` for `pdorg2l`

`COMPLEX` for `pcung2l`

`COMPLEX*16` for `pzung2l`.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, the *j*-th column must contain the vector that defines the elementary reflector *H*`(j)`, $ja+n-k \leq j \leq ja+n-k$, as returned by `p?geqlf` in the *k* columns of its *distributed matrix* argument *A*`(ia:*,ja+n-k:ja+n-1)`.

ia

(global) `INTEGER`.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja

(global) `INTEGER`.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

REAL for psorg2l

DOUBLE PRECISION for pdorg2l

COMPLEX for pcung2l

COMPLEX*16 for pzung2l.

Array of size *LOCc*(*ja*+*n*-1).

tau(*j*) contains the scalar factor of the elementary reflector *H*(*j*), as returned by p?geqlf.

work (local)

REAL for psorg2l

DOUBLE PRECISION for pdorg2l

COMPLEX for pcung2l

COMPLEX*16 for pzung2l.

Workspace array of size *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork* ≥ *mpa*0 + max(1, *nqa*0), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*).

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local)INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?org2r/p?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqrf *(unblocked algorithm).*

Syntax

```
call psorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?org2r/p?ung2rroutine generates an *m*-by-*n* real/complex matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal columns, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*:

Q = *H*(1)**H*(2)*...**H*(*k*)

as returned by p?geqrf.

Input Parameters

m (global) INTEGER.

The number of rows in the distributed submatrix *Q*.*m* ≥ 0.

n (global) INTEGER.

The number of columns in the distributed submatrix *Q*. *m* ≥ *n* ≥ 0.

k (global) INTEGER.

The number of elementary reflectors whose product defines the matrix *Q*. *n* ≥ *k* ≥ 0.

a REAL for psorg2r

DOUBLE PRECISION for pdorg2r

COMPLEX for pcung2r

COMPLEX*16 for pzung2r.

Pointer into the local memory to an array of size(*lld_a*, *LOCc*(*ja*+*n*-1))

On entry, the *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja* ≤ *j* ≤ *ja*+*k*−1, as returned by p?geqrf in the *k* columns of its *distributed matrix* argument *A*(*ia*:*,* *ja*:*ja*+*k*−1).

ia

(global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja

(global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

tau

(local)

REAL for psorg2r

DOUBLE PRECISION for pdorg2r

COMPLEX for pcung2r

COMPLEX*16 for pzung2r.

Array of size *LOCc*(*ja*+*k*-1).

tau(*j*) contains the scalar factor of the elementary reflector *H*(*j*), as returned by p?geqrf. This array is tied to the distributed matrix *A*.

work

(local)

REAL for psorg2r

DOUBLE PRECISION for pdorg2r

COMPLEX for pcung2r

COMPLEX*16 for pzung2r.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least *lwork* ≥ *mpa*0 + max(1, *nqa*0), where

iroffa = mod(*ia*-1, *mb_a* , *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*).

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work — On exit, *work*(1) returns the minimal and optimal *lwork*.

info — (local)INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orgl2/p?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf *(unblocked algorithm).*

Syntax

```
call psorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?orgl2/p?ungl2routine generates a *m*-by-*n* real/complex matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja* +*n*-1) with orthonormal rows, which is defined as the first *m* rows of a product of *k* elementary reflectors of order *n*

Q = *H*(*k*)*...**H*(2)**H*(1) (for real flavors),

Q = $(H(k))^H$*...*$(H(2))^H$*$(H(1))^H$ (for complex flavors) as returned by p?gelqf.

Input Parameters

m — (global) INTEGER.

The number of rows in the distributed submatrix *Q*. *m* ≥ 0.

n — (global) INTEGER.

The number of columns in the distributed submatrix *Q*. *n* ≥ *m* ≥ 0.

k (global) INTEGER.

The number of elementary reflectors whose product defines the matrix *Q*. $m \geq k \geq 0$.

a REAL for psorgl2

DOUBLE PRECISION for pdorgl2

COMPLEX for pcungl2

COMPLEX*16 for pzungl2.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(ja+n-1)).

On entry, the *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the *k* rows of its *distributed matrix* argument *A*(*ia*:*ia*+*k*-1, *ja*:*).

ia (global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja (global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

REAL for psorgl2

DOUBLE PRECISION for pdorgl2

COMPLEX for pcungl2

COMPLEX*16 for pzungl2.

Array of size *LOCr*(ja+k-1). *tau*(*j*) contains the scalar factor of the elementary reflectors *H*(*j*), as returned by p?gelqf. This array is tied to the distributed matrix *A*.

WORK (local)

REAL for psorgl2

DOUBLE PRECISION for pdorgl2

COMPLEX for pcungl2

COMPLEX*16 for pzungl2.

Workspace array of size *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least $lwork \geq nqa0 + \max(1, mpa0)$, where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

mpa0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),

nqa0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*).

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow,* and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work

On exit, *work*(1) returns the minimal and optimal *lwork*.

info

(local)INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orgr2/p?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by p?gerqf *(unblocked algorithm).*

Syntax

```
call psorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?orgr2/p?ungr2routine generates an *m*-by-*n* real/complex matrix *Q* denoting *A*(*ia*:*ia*+*m*–1, *ja*:*ja*+*n*–1) with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *n*

Q = *H*(1)**H*(2)*...**H*(*k*) (for real flavors);

Q = $(H(1))^H$*$(H(2))^H$...*$(H(k))^H$ (for complex flavors) as returned by p?gerqf.

Input Parameters

m

(global) INTEGER.

The number of rows in the distributed submatrix *Q*. $m \geq 0$.

n

(global) INTEGER.

The number of columns in the distributed submatrix *Q*. $n \geq m \geq 0$.

k

(global) INTEGER.

The number of elementary reflectors whose product defines the matrix *Q*. $m \geq k \geq 0$.

a

REAL for psorgr2

DOUBLE PRECISION for pdorgr2

COMPLEX for pcungr2

COMPLEX*16 for pzungr2.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, the *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), $ia+m-k \leq i \leq ia+m-1$, as returned by p?gerqf in the *k* rows of its *distributed matrix* argument *A*(*ia*+*m*-*k*:*ia*+*m*-1, *ja*:*).

ia

(global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja

(global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau

(local)

REAL for psorgr2

DOUBLE PRECISION for pdorgr2

COMPLEX for pcungr2

COMPLEX*16 for pzungr2.

Array of size *LOCr*(*ja*+*m*-1). *tau*(*j*) contains the scalar factor of the elementary reflectors *H*(*j*), as returned by p?gerqf. This array is tied to the distributed matrix *A*.

work

(local)

REAL for psorgr2

DOUBLE PRECISION for pdorgr2

COMPLEX for pcungr2

COMPLEX*16 for pzungr2.

Workspace array of size *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least $lwork \geq nqa0 + \max(1, mpa0)$, where $iroffa = \mathrm{mod}(ia-1, mb\_a)$, $icoffa = \mathrm{mod}(ja-1, nb\_a)$,

$iarow = \mathrm{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow)$,

$iacol = \mathrm{indxg2p}(ja, nb\_a, mycol, csrc\_a, npcol)$,

$mpa0 = \mathrm{numroc}(m+iroffa, mb\_a, myrow, iarow, nprow)$,

$nqa0 = \mathrm{numroc}(n+icoffa, nb\_a, mycol, iacol, npcol)$.

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work On exit, $work(1)$ returns the minimal and optimal *lwork*.

info (local)INTEGER.

$= 0$: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then $info = -(i*100 + j)$,

if the *i*-th argument is a scalar and had an illegal value,

then $info = -i$.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orm2l/p?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf *(unblocked algorithm).*

Syntax

call psorm2l(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdorm2l(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pcunm2l(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunm2l(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The p?orm2l/p?unm2lroutine overwrites the general real/complex *m*-by-*n* distributed matrix sub(*C*)=*C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

*Q**sub(*C*) if *side* = 'L' and *trans* = 'N', or

Q^T*sub(*C*) / Q^H*sub(*C*) if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

sub(*C*)**Q* if *side* = '*R*' and *trans* = 'N', or

sub(*C*)*Q^T / sub(*C*)*Q^H if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

where *Q* is a real orthogonal or complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(*k*)*...**H*(2)**H*(1) as returned by p?geqlf . *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side
(global) CHARACTER.
= 'L': apply *Q* or Q^T for real flavors (Q^H for complex flavors) from the left,
= 'R': apply *Q* or Q^T for real flavors (Q^H for complex flavors) from the right.

trans
(global) CHARACTER.
= 'N': apply *Q* (no transpose)
= 'T': apply Q^T (transpose, for real flavors)
= 'C': apply Q^H (conjugate transpose, for complex flavors)

m
(global) INTEGER.
The number of rows in the distributed matrix sub(*C*). *m* ≥ 0.

n
(global) INTEGER.
The number of columns in the distributed matrix sub(*C*). *n* ≥ 0.

k
(global) INTEGER.
The number of elementary reflectors whose product defines the matrix *Q*.
If *side* = 'L', *m* ≥ *k* ≥ 0;
if *side* = 'R', *n* ≥ *k* ≥ 0.

a
(local)
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*k*-1)).

On entry, the *j*-th row must contain the vector that defines the elementary reflector *H*(*j*), $ja \le j \le ja+k-1$, as returned by p?geqlf in the *k* columns of its distributed matrix argument *A*(*ia*:*,*,*ja*:*ja*+*k*-1). The argument *A*(*ia*:*,*,*ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

If *side* = 'L', lld_a ≥ max(1, *LOCr*(*ia*+*m*-1)),

if *side* = 'R', lld_a ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia (global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja (global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

REAL for psorm2l

DOUBLE PRECISION for pdorm2l

COMPLEX for pcunm2l

COMPLEX*16 for pzunm2l.

Array of size *LOCc*(*ja*+*n*-1). *tau*(*j*) contains the scalar factor of the elementary reflector *H*(*j*), as returned by p?geqlf. This array is tied to the distributed matrix *A*.

c (local)

REAL for psorm2l

DOUBLE PRECISION for pdorm2l

COMPLEX for pcunm2l

COMPLEX*16 for pzunm2l.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc* +*n*-1)).On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) INTEGER.

The row index in the global matrix *C* indicating the first row of sub(*C*).

jc (global) INTEGER.

The column index in the global matrix *C* indicating the first column of sub(*C*).

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)

REAL for psorm2l

DOUBLE PRECISION for pdorm2l

COMPLEX for pcunm2l

COMPLEX*16 for pzunm2l.

Workspace array of size *lwork*.

On exit, *work*(1) returns the minimal and optimal *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least

if *side* = 'L', *lwork* ≥ *mpc*0 + max(1, *nqc*0),

if *side* = 'R', *lwork* ≥ *nqc*0 + max(max(1, *mpc*0), numroc(numroc(*n* +*icoffc*, *nb_a*, 0, 0, *npcol*), *nb_a*, 0, 0, *lcmq*)),

where

lcmq = *lcm*/*npcol*,

lcm = iclm(*nprow*, *npcol*),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = mod(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

Mqc0 = numroc(*m*+*icoffc*, *nb_c*, *mycol*, *icrow*, *nprow*),

Npc0 = numroc(*n*+*iroffc*, *mb_c*, *myrow*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c On exit, *c* is overwritten by *Q**sub(*C*), or Q^T*sub(*C*)/ Q^H*sub(*C*), or sub(*C*)**Q*, or sub(*C*)*Q^T / sub(*C*)*Q^H

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

NOTE
The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*mb_a*.eq.*mb_c* .AND. *iroffa*.eq.*iroffc* .AND. *iarow*.eq.*icrow*)

If *side* = 'R', (*mb_a*.eq.*nb_c* .AND. *iroffa*.eq.*iroffc*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orm2r/p?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?geqrf *(unblocked algorithm).*

Syntax

call psorm2r(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdorm2r(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pcunm2r(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunm2r(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The p?orm2r/p?unm2rroutine overwrites the general real/complex *m*-by-*n* distributed matrix sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) with

*Q**sub(*C*) if *side* = 'L' and *trans* = 'N', or

Q^T*sub(*C*) / Q^H*sub(*C*) if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

sub(*C*)**Q* if *side* = '*R*' and *trans* = 'N', or

sub(*C*)*Q^T / sub(*C*)*Q^H if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = H(*k*)*...**H*(2)**H*(1) as returned by p?geqrf . *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER.

= 'L': apply *Q* or Q^T for real flavors (Q^H for complex flavors) from the left,

= 'R': apply *Q* or Q^T for real flavors (Q^H for complex flavors) from the right.

trans (global) CHARACTER.

= 'N': apply Q (no transpose)

= 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H (conjugate transpose, for complex flavors)

m (global) INTEGER.

The number of rows in the distributed matrix sub(C). $m \geq 0$.

n (global) INTEGER.

The number of columns in the distributed matrix sub(C). $n \geq 0$.

k (global) INTEGER.

The number of elementary reflectors whose product defines the matrix Q.

If *side* = 'L', $m \geq k \geq 0$;

if *side* = 'R', $n \geq k \geq 0$.

a (local)

REAL for psorm2r

DOUBLE PRECISION for pdorm2r

COMPLEX for pcunm2r

COMPLEX*16 for pzunm2r.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*k*-1)).

On entry, the j-th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. The argument $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit.

If *side* = 'L', lld_a ≥ max(1, *LOCr*(*ia*+*m*-1)),

if *side* = 'R', lld_a ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia (global) INTEGER.

The row index in the global matrix A indicating the first row of sub(A).

ja (global) INTEGER.

The column index in the global matrix A indicating the first column of sub(A).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A.

tau (local)

REAL for psorm2r

DOUBLE PRECISION for pdorm2r

COMPLEX for pcunm2r

COMPLEX*16 for pzunm2r.

Array of size *LOCc*(*ja*+*k*-1). *tau*(*j*) contains the scalar factor of the elementary reflector *H*(*j*), as returned by p?geqrf. This array is tied to the distributed matrix *A*.

c

(local)

REAL for psorm2r

DOUBLE PRECISION for pdorm2r

COMPLEX for pcunm2r

COMPLEX*16 for pzunm2r.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc*+*n*-1)).

On entry, the local pieces of the distributed matrix sub (*C*).

ic

(global) INTEGER.

The row index in the global matrix *C* indicating the first row of sub(*C*).

jc

(global) INTEGER.

The column index in the global matrix *C* indicating the first column of sub(*C*).

descc

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *C*.

work

(local)

REAL for psorm2r

DOUBLE PRECISION for pdorm2r

COMPLEX for pcunm2r

COMPLEX*16 for pzunm2r.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least

if *side* = 'L', *lwork* ≥ *mpc*0 + max(1, *nqc*0),

if *side* = 'R', *lwork* ≥ *nqc*0 + max(max(1, *mpc*0), numroc(numroc(*n* +*icoffc*, *nb_a*, 0, 0, *npcol*), *nb_a*, 0, 0, *lcmq*)),

where

lcmq = *lcm*/*npcol* ,

lcm = *iclm*(*nprow*, *npcol*),

iroffc = mod(*ic*-*1*, *mb_c*),

icoffc = mod(*jc*-*1*, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

Mqc0 = numroc(*m*+*icoffc*, *nb_c*, *mycol*, *icrow*, *nprow*),

Npc0 = numroc(*n*+*iroffc*, *mb_c*, *myrow*, *iccol*, *npcol*),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c

On exit, *c* is overwritten by Q*sub(C), or Q^T*sub(C)/ Q^H*sub(C), or sub(C)*Q, or sub(C)*Q^T / sub(C)*Q^H

work

On exit, *work*(1) returns the minimal and optimal *lwork*.

info

(local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

NOTE
The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*–1, *jc*:*jc*+*n*–1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*mb_a*.eq.*mb_c* .AND. *iroffa*.eq.*iroffc* .AND. *iarow*.eq.*icrow*).

If *side* = 'R', (*mb_a*.eq.*nb_c* .AND. *iroffa*.eq.*iroffc*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orml2/p?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf *(unblocked algorithm).*

Syntax

call psorml2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdorml2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pcunml2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunml2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The p?orml2/p?unml2routine overwrites the general real/complex *m*-by-*n* distributed matrix sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) with

*Q**sub(*C*) if *side* = 'L' and *trans* = 'N', or

Q^T*sub(*C*) / Q^H*sub(*C*) if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

sub(*C*)**Q* if *side* = 'R' and *trans* = 'N', or

sub(*C*)*Q^T / sub(*C*)*Q^H if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

where *Q* is a real orthogonal or complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = H(*k*)*...**H*(2)**H*(1) (for real flavors)

Q = (*H*(*k*))^H*...*(*H*(2))^H*(*H*(1))^H (for complex flavors)

as returned by p?gelqf . *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER.

= 'L': apply *Q* or Q^T for real flavors (Q^H for complex flavors) from the left,

= 'R': apply *Q* or Q^T for real flavors (Q^H for complex flavors) from the right.

trans (global) CHARACTER.

= 'N': apply *Q* (no transpose)

= 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H (conjugate transpose, for complex flavors)

m (global) INTEGER.

The number of rows in the distributed matrix sub(*C*). *m* ≥ 0.

n (global) INTEGER.

The number of columns in the distributed matrix sub(*C*). *n* ≥ 0.

k (global) INTEGER.

The number of elementary reflectors whose product defines the matrix *Q*.

If *side* = 'L', *m* ≥ *k* ≥ 0;

if *side* = 'R', *n* ≥ *k* ≥ 0.

a (local)

REAL for psorml2

DOUBLE PRECISION for pdorml2

COMPLEX for pcunml2

COMPLEX*16 for pzunml2.

Pointer into the local memory to an array of size

(*lld_a*, *LOCc*(*ja*+*m*-1)) if *side*='L',

(*lld_a*, *LOCc*(*ja*+*n*-1)) if *side*='R',

where *lld_a* ≥ max (1, *LOCr*(*ia*+*k*-1)).

On entry, the *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia* ≤ *i* ≤ *ia*+*k*-1, as returned by p?gelqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1, *ja*:*). The argument *A*(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia (global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja (global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)

REAL for psorml2

DOUBLE PRECISION for pdorml2

COMPLEX for pcunml2

COMPLEX*16 for pzunml2.

Array of size *LOCc*(*ia*+*k*-1). *tau*(*i*) contains the scalar factor of the elementary reflector *H*(*i*), as returned by p?gelqf. This array is tied to the distributed matrix *A*.

c (local)

REAL for psorml2

DOUBLE PRECISION for pdorml2

COMPLEX for pcunml2

COMPLEX*16 for pzunml2.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc*+*n*-1)). On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) INTEGER.

The row index in the global matrix *C* indicating the first row of sub(*C*).

jc (global) INTEGER.

The column index in the global matrix *C* indicating the first column of sub(*C*).

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)

REAL for psorml2

DOUBLE PRECISION for pdormlr2

COMPLEX for pcunmlr2

COMPLEX*16 for pzunmlr2.

Workspace array of size *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least

if *side* = 'L', *lwork* ≥ *mqc0* + max(max(1, *npc0*), numroc(numroc(*m* +*icoffc*, *mb_a*, 0, 0, *nprow*), *mb_a*, 0, 0, *lcmp*)),

if *side* = 'R', *lwork* ≥ *npc0* + max(1, *mqc0*),

where

lcmp = *lcm* / *nprow*,

lcm = *iclm*(*nprow*, *npcol*),

iroffc = mod(*ic-1*, *mb_c*),

icoffc = mod(*jc-1*, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

Mpc0 = numroc(*m*+*icoffc*, *mb_c*, *mycol*, *icrow*, *nprow*),

Nqc0 = numroc(*n*+*iroffc*, *nb_c*, *myrow*, *iccol*, *npcol*),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c On exit, *c* is overwritten by Q*sub(C), or Q^T*sub(C)/ Q^H*sub(C), or sub(C)*Q, or sub(C)*Q^T / sub(C)*Q^H

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

NOTE
The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*nb_a*.eq.*mb_c* .AND. *icoffa*.eq.*iroffc*)

If *side* = 'R', (*nb_a*.eq.*nb_c* .AND. *icoffa*.eq.*icoffc* .AND. *iacol*.eq.*iccol*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormr2/p?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf *(unblocked algorithm).*

Syntax

call psormr2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdormr2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pcunmr2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmr2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The p?ormr2/p?unmr2routine overwrites the general real/complex *m*-by-*n* distributed matrix sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) with

*Q**sub(*C*) if *side* = 'L' and *trans* = 'N', or

Q^T*sub(*C*) / Q^H*sub(*C*) if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

sub(*C*)**Q* if *side* = 'R' and *trans* = 'N', or

sub(*C*)*Q^T / sub(*C*)*Q^H if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

where *Q* is a real orthogonal or complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1)**H*(2)*...**H*(*k*) (for real flavors)

Q = $(H(1))^H$*$(H(2))^H$*...*$(H(k))^H$ (for complex flavors)

as returned by p?gerqf . *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER.

= 'L': apply *Q* or Q^T for real flavors (Q^H for complex flavors) from the left,

= 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.

trans

(global) CHARACTER.

= 'N': apply Q (no transpose)

= 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H(conjugate transpose, for complex flavors)

m

(global) INTEGER.

The number of rows in the distributed matrix sub(C). $m \geq 0$.

n

(global) INTEGER.

The number of columns in the distributed matrix sub(C). $n \geq 0$.

k

(global) INTEGER.

The number of elementary reflectors whose product defines the matrix Q.

If *side* = 'L', $m \geq k \geq 0$;

if *side* = 'R', $n \geq k \geq 0$.

a

(local)

REAL for psormr2

DOUBLE PRECISION for pdormr2

COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.

Pointer into the local memory to an array of size

(*lld_a*, *LOCc*(*ja*+*m*-1)) if *side*='L',

(*lld_a*, *LOCc*(*ja*+*n*-1)) if *side*='R',

where *lld_a* ≥ max (1, *LOCr*(*ia*+*k*-1)).

On entry, the *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia* ≤ *i* ≤ *ia*+*k*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1, *ja*:*).

The argument *A*(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia

(global) INTEGER.

The row index in the global matrix *A* indicating the first row of sub(*A*).

ja

(global) INTEGER.

The column index in the global matrix *A* indicating the first column of sub(*A*).

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau

(local)

REAL for psormr2

DOUBLE PRECISION for pdormr2

COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.

Array of size *LOCc*(*ia*+*k*−1). *tau*(*j*) contains the scalar factor of the elementary reflector *H*(*j*), as returned by p?gerqf. This array is tied to the distributed matrix *A*.

c (local)

REAL for psormr2

DOUBLE PRECISION for pdormr2

COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.

Pointer into the local memory to an array of size (*lld_c*, *LOCc*(*jc*+*n*−1)). On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) INTEGER.

The row index in the global matrix *C* indicating the first row of sub(*C*).

jc (global) INTEGER.

The column index in the global matrix *C* indicating the first column of sub(*C*).

descc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)

REAL for psormr2

DOUBLE PRECISION for pdormr2

COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.

Workspace array of size *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least

if *side* = 'L', *lwork* ≥ *mpc*0 + max(max(1, *nqc*0), numroc(numroc(*m* +*iroffc*, *mb_a*, 0, 0, *nprow*), *mb_a*, 0, 0, *lcmp*)),

if *side* = 'R', *lwork* ≥ *nqc*0 + max(1, *mpc*0),

where *lcmp* = *lcm*/*nprow*,

lcm = iclm(*nprow*, *npcol*),

iroffc = mod(*ic*-*1*, *mb_c*),

icoffc = mod(*jc*-*1*, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

Mpc0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

Nqc0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — On exit, *c* is overwritten by Q*sub(C), or Q^T*sub(C)/ Q^H*sub(C), or sub(C)*Q, or sub(C)*Q^T / sub(C)*Q^H

work — On exit, *work*(1) returns the minimal and optimal *lwork*.

info — (local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

NOTE
The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*nb_a*.eq.*mb_c* .AND. *icoffa*.eq.*iroffc*).

If *side* = 'R', (*nb_a*.eq.*nb_c* .AND. *icoffa*.eq.*icoffc* .AND. *iacol*.eq.*iccol*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?pbtrsv

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf.

Syntax

call pspbtrsv(*uplo*, *trans*, *n*, *bw*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pdpbtrsv(*uplo*, *trans*, *n*, *bw*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pcpbtrsv(*uplo*, *trans*, *n*, *bw*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

```
call pzpbtrsv(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Description

The p?pbtrsvroutine solves a banded triangular system of linear equations

$A(1:n, ja:ja+n-1)*X = B(jb:jb+n-1, 1:nrhs)$

or

$A(1:n, ja:ja+n-1)^T*X = B(jb:jb+n-1, 1:nrhs)$ for real flavors,

$A(1:n, ja:ja+n-1)^H*X = B(jb:jb+n-1, 1:nrhs)$ for complex flavors,

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Cholesky factorization code p?pbtrf and is stored in $A(1:n, ja:ja+n-1)$ and *af*. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to *uplo*.

The routine p?pbtrf must be called first.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

uplo

(global) CHARACTER. Must be 'U' or 'L'.

If *uplo* = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored;

If *uplo* = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

trans

(global) CHARACTER. Must be 'N' or 'T' or 'C'.

If *trans* = 'N', solve with $A(1:n, ja:ja+n-1)$;

If *trans* = 'T' or 'C' for real flavors, solve with $A(1:n, ja:ja+n-1)^T$.

If *trans* = 'C' for complex flavors, solve with conjugate transpose $(A(1:n, ja:ja+n-1)^H$.

n

(global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.

bw

(global) INTEGER.

The number of subdiagonals in 'L' or 'U', $0 \leq bw \leq n-1$.

nrhs

(global) INTEGER.

The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.

a

(local)

REAL for pspbtrsv

DOUBLE PRECISION for pdpbtrsv

COMPLEX for pcpbtrsv

COMPLEX*16 for pzpbtrsv.

Pointer into the local memory to an array with the first size *lld_a* ≥ (*bw* +1), stored in *desca*.

On entry, this array contains the local pieces of the *n*-by-*n* symmetric banded distributed Cholesky factor *L* or $L^T*A(1:n, ja:ja+n-1)$.

This local portion is stored in the packed banded format used in LAPACK. See the *Application Notes* below and the ScaLAPACK manual for more detail on the format of distributed matrices.

ja (global) INTEGER. The index in the global in the global matrix *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If 1D type (*dtype_a* = 501), then *dlen*≥ 7;

If 2D type (*dtype_a* = 1), then *dlen*≥ 9.

Contains information on mapping of *A* to memory. (See ScaLAPACK manual for full description and options.)

b (local)

REAL for pspbtrsv

DOUBLE PRECISION for pdpbtrsv

COMPLEX for pcpbtrsv

COMPLEX*16 for pzpbtrsv.

Pointer into the local memory to an array of local lead size *lld_b*≥*nb*.

On entry, this array contains the local pieces of the right hand sides *B*(*jb*:*jb*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global matrix *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

If 1D type (*dtype_b* = 502), then *dlen*≥ 7;

If 2D type (*dtype_b* = 1), then *dlen*≥ 9.

Contains information on mapping of *B* to memory. Please, see ScaLAPACK manual for full description and options.

laf (local)

INTEGER. The size of user-input auxiliary fill-in space *af*. Must be *laf* ≥ (*nb*+2***bw*)***bw* . If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local)

REAL for pspbtrsv

DOUBLE PRECISION for pdpbtrsv

COMPLEX for pcpbtrsv

COMPLEX*16 for pzpbtrsv.

The array *work* is a temporary workspace array of size *lwork*. This space may be overwritten in between calls to routines.

lwork (local or global) INTEGER. The size of the user-input workspace *work*, must be at least *lwork* ≥ *bw***nrhs*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

af (local)

REAL for pspbtrsv

DOUBLE PRECISION for pdpbtrsv

COMPLEX for pcpbtrsv

COMPLEX*16 for pzpbtrsv.

The array *af* is of size *laf*. It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?pbtrf and is stored in *af*. If a linear system is to be solved using p?pbtrs after the factorization routine, *af* must not be altered after the factorization.

b On exit, this array contains the local piece of the solutions distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork*.

info (local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

Application Notes

If the factorization routine and the solve routine are to be called separately to solve various sets of right-hand sides using the same coefficient matrix, the auxiliary space *af* must not be altered between calls to the factorization routine and the solve routine.

The best algorithm for solving banded and tridiagonal linear systems depends on a variety of parameters, especially the bandwidth. Currently, only algorithms designed for the case N/P>>*bw* are implemented. These algorithms go by many names, including Divide and Conquer, Partitioning, domain decomposition-type, etc.

The Divide and Conquer algorithm assumes the matrix is narrowly banded compared with the number of equations. In this situation, it is best to distribute the input matrix *A* one-dimensionally, with columns atomic and rows divided amongst the processes. The basic algorithm divides the banded matrix up into *P* pieces with one stored on each processor, and then proceeds in 2 phases for the factorization or 3 for the solution of a linear system.

1. **Local Phase**: The individual pieces are factored independently and in parallel. These factors are applied to the matrix creating fill-in, which is stored in a non-inspectable way in auxiliary space *af*. Mathematically, this is equivalent to reordering the matrix *A* as PAP^T and then factoring the principal leading submatrix of size equal to the sum of the sizes of the matrices factored on each processor. The factors of these submatrices overwrite the corresponding parts of *A* in memory.
2. **Reduced System Phase**: A small `(bw*(P-1))` system is formed representing interaction of the larger blocks and is stored (as are its factors) in the space `af`. A parallel Block Cyclic Reduction algorithm is used. For a linear system, a parallel front solve followed by an analogous backsolve, both using the structure of the factored matrix, are performed.
3. **Back Subsitution Phase**: For a linear system, a local backsubstitution is performed on each processor in parallel.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?pttrsv

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by `p?pttrf` *.*

Syntax

```
call pspttrsv(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pdpttrsv(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcpttrsv(uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzpttrsv(uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Description

The `p?pttrsv`routine solves a tridiagonal triangular system of linear equations

$A(1:n, ja:ja+n-1)*X = B(jb:jb+n-1, 1:nrhs)$

or

$A(1:n, ja:ja+n-1)^T*X = B(jb:jb+n-1, 1:nrhs)$ for real flavors,

$A(1:n, ja:ja+n-1)^H*X = B(jb:jb+n-1, 1:nrhs)$ for complex flavors,

where $A(1:n, ja:ja+n-1)$ is a tridiagonal triangular matrix factor produced by the Cholesky factorization code `p?pttrf` and is stored in $A(1:n, ja:ja+n-1)$ and `af`. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to `uplo`.

The routine `p?pttrf` must be called first.

Input Parameters

`uplo` (global) `CHARACTER`. Must be `'U'` or `'L'`.

If `uplo = 'U'`, upper triangle of $A(1:n, ja:ja+n-1)$ is stored;

If *uplo* = 'L', lower triangle of *A*(1:*n*, *ja*:*ja*+*n*-1) is stored.

trans

(global) CHARACTER. Must be 'N' or 'C'.

If *trans* = 'N', solve with *A*(1:*n*, *ja*:*ja*+*n*-1);

If *trans* = 'C' (for complex flavors), solve with conjugate transpose $(A(1:n, ja:ja+n-1))^H$.

n

(global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed submatrix *A*(1:*n*, *ja*:*ja*+*n*-1). $n \geq 0$.

nrhs

(global) INTEGER.

The number of right hand sides; the number of columns of the distributed submatrix *B*(*jb*:*jb*+*n*-1, 1:*nrhs*); $nrhs \geq 0$.

d

(local)

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

Pointer to the local part of the global vector storing the main diagonal of the matrix; must be of size ≥*nb_a*.

e

(local)

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

Pointer to the local part of the global vector *du* storing the upper diagonal of the matrix; must be of size ≥*nb_a*. Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja

(global) INTEGER. The index in the global matrix *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If 1D type (*dtype_a* = 501 or 502), then $dlen \geq 7$;

If 2D type (*dtype_a* = 1), then $dlen \geq 9$.

Contains information on mapping of *A* to memory. See ScaLAPACK manual for full description and options.

b

(local)

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

Pointer into the local memory to an array of local lead size *lld_b* ≥ *nb*.

On entry, this array contains the local pieces of the right hand sides *B*(*jb*:*jb*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global matrix *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

If 1D type (*dtype_b* = 502), then *dlen* ≥ 7;

If 2D type (*dtype_b* = 1), then *dlen* ≥ 9.

Contains information on mapping of *B* to memory. See ScaLAPACK manual for full description and options.

laf (local)

INTEGER. The size of user-input auxiliary fill-in space *af*. Must be *laf* ≥ (*nb*+2**bw*)**bw*.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local)

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

The array *work* is a temporary workspace array of size *lwork*. This space may be overwritten in between calls to routines.

lwork (local or global) INTEGER. The size of the user-input workspace *work*, must be at least *lwork*≥(10+2*min(100, *nrhs*))***npcol*+4***nrhs*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

d, *e* (local).

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

On exit, these arrays contain information on the factors of the matrix.

af (local)

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

The array *af* is of size *laf*. It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?pbtrf and is stored in *af*. If a linear system is to be solved using p?pttrs after the factorization routine, *af* must not be altered after the factorization.

b — On exit, this array contains the local piece of the solutions distributed matrix *X*.

work(1) — On exit, *work*(1) contains the minimum value of *lwork*.

info — (local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?potf2

Computes the Cholesky factorization of a symmetric/ Hermitian positive definite matrix (local unblocked algorithm).

Syntax

```
call pspotf2(uplo, n, a, ia, ja, desca, info)
call pdpotf2(uplo, n, a, ia, ja, desca, info)
call pcpotf2(uplo, n, a, ia, ja, desca, info)
call pzpotf2(uplo, n, a, ia, ja, desca, info)
```

Description

The p?potf2routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite distributed matrix sub (*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

The factorization has the form

sub(*A*) = *U*'**U*, if *uplo* = 'U', or sub(*A*) = *L***L*', if *uplo* = 'L',

where *U* is an upper triangular matrix, *L* is lower triangular. *X*' denotes transpose (conjugate transpose) of *X*.

Input Parameters

uplo — (global) CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored.

= 'U': upper triangle of sub (*A*) is stored;

= 'L': lower triangle of sub (*A*) is stored.

n

(global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed matrix sub (*A*). *n* ≥ 0.

a

(local)

REAL for pspotf2

DOUBLE PRECISION for pdpotf2

COMPLEX for pcpotf2

COMPLEX*16 for pzpotf2.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*) to be factored.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix and the strictly lower triangular part of this matrix is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a

(local)

On exit,

if *uplo* = 'U', the upper triangular part of the distributed matrix contains the Cholesky factor U;

if *uplo* = 'L', the lower triangular part of the distributed matrix contains the Cholesky factor L.

info

(local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100 +*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?rot

Applies a planar rotation to two distributed vectors.

Syntax

```
call psrot( n, x, ix, jx, descx, incx, y, iy, jy, descy, incy, cs, sn, work, lwork, info )
call pdrot( n, x, ix, jx, descx, incx, y, iy, jy, descy, incy, cs, sn, work, lwork, info )
```

Description

p?rot applies a planar rotation defined by *cs* and *sn* to the two distributed vectors sub(*x*) and sub(*y*).

Input Parameters

n (global) INTEGER

The number of elements to operate on when applying the planar rotation to *x* and *y* (*n*≥0).

x REAL for psrot

DOUBLE PRECISION for pdrot

(local) array of size ((*jx*-1)*m_x + *ix* + (*n* - 1)*abs(*incx*))

This array contains the entries of the distributed vector sub(*x*).

ix (global) INTEGER

The global row index of the submatrix of the distributed matrix *x* to operate on. If *incx* = 1, then it is required that *ix* = *iy*. 1 ≤*ix*≤*m_x*.

jx (global) INTEGER

The global column index of the submatrix of the distributed matrix *x* to operate on. If *incx* = *m_x*, then it is required that *jx* = *jy*. 1 ≤*ix*≤*n_x*.

descx (global and local) INTEGER array of size 9

The array descriptor of the distributed matrix *x*.

incx (global) INTEGER

The global increment for the elements of *x*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. Moreover, it must hold that *incx* = *m_x* if *incy* =*m_y* and that *incx* = 1 if *incy* = 1.

y REAL for psrot

DOUBLE PRECISION for pdrot

(local) array of size ((*jy*-1)*m_y + *iy* + (*n* - 1)*abs(*incy*))

This array contains the entries of the distributed vector sub(*y*).

iy

(global) INTEGER

The global row index of the submatrix of the distributed matrix *y* to operate on. If *incy* = 1, then it is required that *iy* = *ix*. 1 ≤*iy*≤*m_y*.

jy

(global) INTEGER

The global column index of the submatrix of the distributed matrix *y* to operate on. If *incy* = *m_x*, then it is required that *jy* = *jx*. 1 ≤*jy*≤*m_y*.

descy

(global and local) INTEGER array of size 9

The array descriptor of the distributed matrix *y*.

incy

(global) INTEGER

The global increment for the elements of *y*. Only two values of *incy* are supported in this version, namely 1 and *m_y*. Moreover, it must hold that *incy* = *m_y* if *incx* = *m_x* and that *incy* = 1 if *incx* = 1.

cs, *sn*

(global)

REAL for psrot

DOUBLE PRECISION for pdrot

The parameters defining the properties of the planar rotation. It must hold that 0 ≤*cs*,*sn*≤ 1 and that $sn^2 + cs^2 = 1$. The latter is hardly checked in finite precision arithmetics.

work

REAL for psrot

DOUBLE PRECISION for pdrot

(local workspace) array of size *lwork*

lwork

(local) INTEGER

The length of the workspace array *work*.

If *incx* = 1 and *incy* = 1, then *lwork* = 2**m_x*

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by pxerbla.

OUTPUT Parameters

x

y

work(1)

On exit, if *info* = 0, *work*(1) returns the optimal *lwork*

info

(global) INTEGER

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call psrscl(n, sa, sx, ix, jx, descx, incx)
call pdrscl(n, sa, sx, ix, jx, descx, incx)
call pcsrscl(n, sa, sx, ix, jx, descx, incx)
call pzdrscl(n, sa, sx, ix, jx, descx, incx)
```

Description

The p?rsclroutine multiplies an *n*-element real/complex vector sub(*X*) by the real scalar 1/*a*. This is done without overflow or underflow as long as the final result sub(*X*)/a does not overflow or underflow.

sub(*X*) denotes *X*(*ix*:*ix*+*n*-1, *jx*:*jx*), if *incx* = 1,

and *X*(*ix*:*ix*, *jx*:*jx*+*n*-1), if *incx* = *m_x*.

Input Parameters

n

(global) INTEGER.

The number of components of the distributed vector sub(*X*). *n* ≥ 0.

sa

REAL for psrscl/pcsrscl

DOUBLE PRECISION for pdrscl/pzdrscl.

The scalar *a* that is used to divide each component of the vector sub(*X*). This parameter must be ≥ 0.

sx

REAL forpsrscl

DOUBLE PRECISION for pdrscl

COMPLEX for pcsrscl

COMPLEX*16 for pzdrscl.

Array containing the local pieces of a distributed matrix of size of at least ((*jx*-1)**m_x* + *ix* + (*n*-1)*abs(*incx*)). This array contains the entries of the distributed vector sub(*X*).

ix

(global) INTEGER.The row index of the submatrix of the distributed matrix *X* to operate on.

jx

(global) INTEGER.

The column index of the submatrix of the distributed matrix *X* to operate on.

descx

(global and local) INTEGER.

Array of size 9. The array descriptor for the distributed matrix *X*.

incx (global) INTEGER.

The increment for the elements of *X*. This version supports only two values of *incx*, namely 1 and *m_x*.

Output Parameters

sx On exit, the result *x*/*a*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?sygs2/p?hegs2

Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf *(local unblocked algorithm).*

Syntax

```
call pssygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pdsygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pchegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pzhegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

The p?sygs2/p?hegs2routine reduces a real symmetric-definite or a complex Hermitian positive-definite generalized eigenproblem to standard form.

Here sub(*A*) denotes *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), and sub(*B*) denotes *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1).

If *ibtype* = 1, the problem is

```
sub(A)*x = λ*sub(B)*x
```

and sub(*A*) is overwritten by

inv(U^T)*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(L^T) - for real flavors, and

inv(U^H)*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(L^H) - for complex flavors.

If *ibtype* = 2 or 3, the problem is

```
sub(A)*sub(B)x = λ*x or sub(B)*sub(A)x =λ*x
```

and sub(*A*) is overwritten by

*U*sub(*A*)*U^T or *L****T*sub(*A*)**L*- for real flavors and

*U*sub(*A*)*U^H or *L****H*sub(*A*)**L*- for complex flavors.

The matrix sub(*B*) must have been previously factorized as U^T*U or *L*L^T (for real flavors), or as U^H*U or *L*L^H (for complex flavors) by p?potrf.

Input Parameters

ibtype (global) INTEGER.

= 1:

compute inv(U^T)*sub(A)*inv(U), or inv(L)*sub(A)*inv(L^T) for real subroutines,

and inv(U^H)*sub(A)*inv(U), or inv(L)*sub(A)*inv(L^H) for complex subroutines;

= 2 or 3:

compute U*sub(A)*U^T, or L^T*sub(A)*L for real subroutines,

and U*sub(A)*U^H or L^H*sub(A)*L for complex subroutines.

uplo (global) CHARACTER

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(*A*) is stored, and how sub(*B*) is factorized.

= 'U': Upper triangular of sub(*A*) is stored and sub(*B*) is factorized as U^T*U (for real subroutines) or as U^H*U (for complex subroutines).

= 'L': Lower triangular of sub(*A*) is stored and sub(*B*) is factorized as $L*L^T$ (for real subroutines) or as $L*L^H$ (for complex subroutines)

n (global) INTEGER.

The order of the matrices sub(*A*) and sub(*B*). $n \geq 0$.

a (local)

REAL for pssygs2

DOUBLE PRECISION for pdsygs2

COMPLEX for pchegs2

COMPLEX*16 for pzhegs2.

Pointer into the local memory to an array of size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(*A*) is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja* (global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the sub(*A*), respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

B (local)

REAL for pssygs2

DOUBLE PRECISION for pdsygs2

COMPLEX for pchegs2

COMPLEX*16 for pzhegs2.

Pointer into the local memory to an array of size (*lld_b*, *LOCc*(*jb*+*n*-1)).

On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of sub(*B*) as returned by p?potrf.

ib, *jb* (global) INTEGER.

The row and column indices in the global matrix *B* indicating the first row and the first column of the sub(*B*), respectively.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Output Parameters

a (local)

On exit, if *info* = 0, the transformed matrix is stored in the same format as sub(*A*).

info INTEGER.

= 0: successful exit.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100+ *j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?sytd2/p?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).

Syntax

```
call pssytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pdsytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pchetd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pzhetd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Description

The p?sytd2/p?hetd2routine reduces a real symmetric/complex Hermitian matrix sub(*A*) to symmetric/ Hermitian tridiagonal form *T* by an orthogonal/unitary similarity transformation:

Q'*sub(*A*)**Q* = *T*, where sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

Input Parameters

uplo

(global) CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(*A*) is stored:

= 'U': upper triangular

= 'L': lower triangular

n

(global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(*A*). $n \geq 0$.

a

(local)

REAL for pssytd2

DOUBLE PRECISION for pdsytd2

COMPLEX for pchetd2

COMPLEX*16 for pzhetd2.

Pointer into the local memory to an array of size $(lld\_a, LOC_c(ja+n-1))$.

On entry, this array contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(*A*) is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work

(local)

REAL for pssytd2

DOUBLE PRECISION for pdsytd2

COMPLEX for pchetd2

COMPLEX*16 for pzhetd2.

The array *work* is a temporary workspace array of size *lwork*.

Output Parameters

a

On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;

if *uplo* = 'L', the diagonal and first subdiagonal of *A* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau,* represent the orthogonal/ unitary matrix *Q* as a product of elementary reflectors. See the *Application Notes* below.

d

(local)

REAL for pssytd2/pchetd2

DOUBLE PRECISION for pdsytd2/pzhetd2.

Array of size*LOCc*(*ja*+*n*-1). The diagonal elements of the tridiagonal matrix *T*:

d(i) = *a*(i,i); *d* is tied to the distributed matrix *A*.

e

(local)

REAL for pssytd2/pchetd2

DOUBLE PRECISION for pdsytd2/pzhetd2.

Array of size *LOCc*(*ja*+*n*-1),

if *uplo* = 'U', *LOCc*(*ja*+*n*-2) otherwise.

The off-diagonal elements of the tridiagonal matrix *T*:

e(i) = *a*(i,i+1) if *uplo* = 'U',

e(i) = *a*(i+1,i) if *uplo* = 'L'.

e is tied to the distributed matrix *A*.

tau

(local)

REAL for pssytd2

DOUBLE PRECISION for pdsytd2

COMPLEX for pchetd2

COMPLEX*16 for pzhetd2.

Array of size *LOCc*(*ja*+*n*-1).

The scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)

On exit, *work*(1) returns the minimal and optimal value of *lwork*.

lwork

(local or global) INTEGER.

The size of the workspace array *work*.

lwork is local input and must be at least *lwork* ≥ 3*n*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

info

(local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = -(*i**100+*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*n*-1)*...**H*(2)**H*(1)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*',

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*i*+1:*n*) = 0 and *v*(*i*) = 1; *v*(1:*i*-1) is stored on exit in *A*(*ia*:*ia*+*i*-2, *ja*+*i*), and *tau* in *tau*(*ja*+*i*-1).

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1)**H*(2)*...**H*(*n*-1).

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*' ,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*) = 0 and *v*(*i*+1) = 1; *v*(*i*+2:*n*) is stored on exit in *A*(*ia*+*i*+1:*ia*+*n*-1, *ja*+*i*-1), and *tau* in *tau*(*ja*+*i*-1).

The contents of sub (*A*) on exit are illustrated by the following examples with *n* = 5:

if *uplo*='U':

$$\begin{bmatrix} d & e & v_2 & v_3 & v_4 \\ & d & e & v_3 & v_4 \\ & & d & e & v_4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

if *uplo*='L':

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v_2 & e & d & \\ v1 & v_2 & v_3 & e & d \end{bmatrix}$$

where *d* and *e* denotes diagonal and off-diagonal elements of *T*, and v_i denotes an element of the vector defining *H*(i).

NOTE

The distributed matrix sub(*A*) must verify some alignment properties, namely the following expression should be true:

(*mb_a*.eq.*nb_a* .AND. *iroffa*.eq.*icoffa*)with *iroffa* = mod(*ia* - 1, *mb_a*) and *icoffa* = mod(*ja* -1, *nb_a*).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trord

Reorders the Schur factorization of a general matrix.

Syntax

```
call pstrord( compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr, wi, m,
work, lwork, iwork, liwork, info )
```

```
call pdtrord( compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr, wi, m,
work, lwork, iwork, liwork, info )
```

Description

`p?trord` reorders the real Schur factorization of a real matrix $A = Q*T*Q^T$, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace.

T must be in Schur form (as returned by `p?lahqr`), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

This subroutine uses a delay and accumulate procedure for performing the off-diagonal updates.

Input Parameters

compq

(global) `CHARACTER*1`

= 'V': update the matrix *q* of Schur vectors;

= 'N': do not update *q*.

select

(global) `INTEGER` array of size *n*

select specifies the eigenvalues in the selected cluster. To select a real eigenvalue *w*(*j*), *select*(*j*) must be set to 1. To select a complex conjugate pair of eigenvalues *w*(*j*) and *w*(*j*+1), corresponding to a 2-by-2 diagonal block, either *select*(*j*) or *select*(*j*+1) or both must be set to 1; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.

para

(global) `INTEGER*6`

Block parameters:

para(1)	maximum number of concurrent computational windows allowed in the algorithm; 0 < *para*(1)≤ min(*nprow*, *npcol*) must hold;
para(2)	number of eigenvalues in each window; 0 < *para*(2) < *para*(3) must hold;
para(3)	window size; *para*(2) < *para*(3) < *mb_t* must hold;
para(4)	minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; 0 ≤*para*(4)≤ 100 must hold;

	para(5)	width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; 0 < *para*(5)≤*mb_t* must hold.
	para(6)	the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size; 0 < *para*(6)≤*para*(2) must hold.

n (global) INTEGER

The order of the globally distributed matrix *t*. *n*≥ 0.

t REAL for pstrord

DOUBLE PRECISION for pdtrord

(local) array of size (*lld_t*,*LOC*$_c$(*n*)).

The local pieces of the global distributed upper quasi-triangular matrix *T*, in Schur form.

it, *jt* (global) INTEGER

The row and column index in the global matrix *T* indicating the first column of *T*. *it* = *jt* = 1 must hold (see Application Notes).

desct (global and local) INTEGER array of size *dlen_*.

The array descriptor for the global distributed matrix *T*.

q REAL for pstrord

DOUBLE PRECISION for pdtrord

(local) array of size (*lld_q*,*LOC*$_c$(*n*)).

On entry, if *compq* = 'V', the local pieces of the global distributed matrix *Q* of Schur vectors.

If *compq* = 'N', *q* is not referenced.

iq, *jq* (global) INTEGER

The column index in the global matrix *Q* indicating the first column of *Q*. *iq* = *jq* = 1 must hold (see Application Notes).

descq (global and local) INTEGER array of size *dlen_*.

The array descriptor for the global distributed matrix *Q*.

work REAL for pstrord

DOUBLE PRECISION for pdtrord

(local workspace) array of size *lwork*

lwork (local) INTEGER

The size of the array *work*.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by pxerbla.

iwork (local workspace) INTEGER array of size *liwork*

liwork (local) INTEGER

The size of the array *iwork*.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by pxerbla

OUTPUT Parameters

select (global) INTEGER array of size *n*

The (partial) reordering is displayed.

t On exit, *t* is overwritten by the local pieces of the reordered matrix *T*, again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.

q On exit, if *compq* = 'V', *q* has been postmultiplied by the global orthogonal transformation matrix which reorders *t*; the leading *m* columns of *q* form an orthonormal basis for the specified invariant subspace.

If *compq* = 'N', *q* is not referenced.

wr, *wi* REAL for pstrord

DOUBLE PRECISION for pdtrord

(global) array of size *n*

The real and imaginary parts, respectively, of the reordered eigenvalues of the matrix *T*. The eigenvalues are in principle stored in the same order as on the diagonal of *T*, with *wr*(*i*) = *t*(*i*,*i*) and, if *t*(*i*:*i*+1,*i*:*i*+1) is a 2-by-2 diagonal block, *wi*(*i*) > 0 and *wi*(*i*+1) = -*wi*(*i*).

Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

m (global) INTEGER

The size of the specified invariant subspace.

$0 \leq m \leq n$.

work(1) On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

iwork(1) On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*.

info (global) INTEGER

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument had an illegal value. If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**1000+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

> 0: here we have several possibilities

- Reordering of *t* failed because some eigenvalues are too close to separate (the problem is very ill-conditioned);

  t may have been partially reordered, and *wr* and *wi* contain the eigenvalues in the same order as in *t*.

  On exit, *info* = {the index of *t* where the swap failed}.
- A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block.

  On exit, *info* = {the index of *t* where the swap failed}.
- If *info* = *n*+1, there is no valid BLACS context (see the BLACS documentation for details).

Application Notes

The following alignment requirements must hold:

- *mb_t* = *nb_t* = *mb_q* = *nb_q*
- *rsrc_t* = *rsrc_q*
- *csrc_t* = *csrc_q*

All matrices must be blocked by a block factor larger than or equal to two (3). This is to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of *t* and *q*, i.e., *it* = *jt* = *iq* = *jq* = 1 must hold. This is however no limitation since p?lahqr does not compute Schur forms of submatrices anyway.

Parallel execution recommendations:

- Use a square grid, if possible, for maximum performance. The block parameters in *para* should be kept well below the data distribution block size.
- In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.

Syntax

```
call pstrsen( job, compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr, wi, m, s, sep, work, lwork, iwork, liwork, info )
```

```
call pdtrsen( job, compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr, wi, m, s, sep, work, lwork, iwork, liwork, info )
```

Description

p?trsen reorders the real Schur factorization of a real matrix $A = Q*T*Q^T$, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace. The reordering is performed by p?trord.

Optionally the routine computes the reciprocal condition numbers of the cluster of eigenvalues and/or the invariant subspace.

T must be in Schur form (as returned by p?lahqr), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

Input Parameters

job

(global) CHARACTER*1

Specifies whether condition numbers are required for the cluster of eigenvalues (*s*) or the invariant subspace (*sep*):

= 'N': no condition numbers are required;

= 'E': only the condition number for the cluster of eigenvalues is computed (*s*);

= 'V': only the condition number for the invariant subspace is computed (*sep*);

= 'B': condition numbers for both the cluster and the invariant subspace are computed (*s* and *sep*).

compq

(global) CHARACTER*1

= 'V': update the matrix *q* of Schur vectors;

= 'N': do not update *q*.

select

(global) LOGICAL array of size *n*

select specifies the eigenvalues in the selected cluster. To select a real eigenvalue *w*(*j*), *select*(*j*) must be set to .TRUE.. To select a complex conjugate pair of eigenvalues *w*(*j*) and *w*(*j*+1), corresponding to a 2-by-2 diagonal block, either *select*(*j*) or *select*(*j*+1) or both must be set to .TRUE.; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.

para

(global) INTEGER*6

Block parameters:

para(1) maximum number of concurrent computational windows allowed in the algorithm; 0 < *para*(1)≤ min(NPROW,NPCOL) must hold;

para(2)	number of eigenvalues in each window; 0 < *para*(2) < *para*(3) must hold;
para(3)	window size; *para*(2) < *para*(3) < *mb_t* must hold;
para(4)	minimal percentage of flops required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; 0 ≤*para*(4)≤ 100 must hold;
para(5)	width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; 0 < *para*(5)≤*mb_t* must hold.
para(6)	the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size 0 < *para*(6)≤*para*(2) must hold;

n

(global) INTEGER

The order of the globally distributed matrix *t*. *n*≥ 0.

t

REAL for pstrsen

DOUBLE PRECISION for pdtrsen

(local) array of size (*lld_t*,*LOC*$_c$(*n*)).

The local pieces of the global distributed upper quasi-triangular matrix *T*, in Schur form.

it, *jt*

(global) INTEGER

The row and column index in the global matrix *T* indicating the first column of *T*. *it* = *jt* = 1 must hold (see Application Notes).

desct

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the global distributed matrix *T*.

q

REAL for pstrsen

DOUBLE PRECISION for pdtrsen

(local) array of size (*lld_q*,*LOC*$_c$(*n*)).

On entry, if *compq* = 'V', the local pieces of the global distributed matrix *Q* of Schur vectors.

If *compq* = 'N', *q* is not referenced.

iq, *jq*

(global) INTEGER

The column index in the global matrix *Q* indicating the first column of *Q*. *iq* = *jq* = 1 must hold (see Application Notes).

descq

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the global distributed matrix *Q*.

work

REAL for pstrsen

DOUBLE PRECISION for pdtrsen

(local workspace) array of size *lwork*

lwork

(local) INTEGER

The size of the array *work*.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by pxerbla.

iwork

(local workspace) INTEGER array of size *liwork*

liwork

(local) INTEGER

The size of the array *iwork*.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by pxerbla.

OUTPUT Parameters

t

t is overwritten by the local pieces of the reordered matrix *T*, again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.

q

On exit, if *compq* = 'V', *q* has been postmultiplied by the global orthogonal transformation matrix which reorders *t*; the leading *m* columns of *q* form an orthonormal basis for the specified invariant subspace.

If *compq* = 'N', *q* is not referenced.

wr, *wi*

REAL for pstrsen

DOUBLE PRECISION for pdtrsen

(global) array of size *n*

The real and imaginary parts, respectively, of the reordered eigenvalues of the matrix *T*. The eigenvalues are in principle stored in the same order as on the diagonal of *T*, with *wr*(*i*) = *t*(*i*,*i*) and, if *t*(*i*:*i*+1,*i*:*i*+1) is a 2-by-2 diagonal block, *wi*(*i*) > 0 and *wi*(*i*+1) = -*wi*(*i*).

Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

m

(global) INTEGER

The size of the specified invariant subspace. 0 ≤*m*≤*n*.

s

REAL for pstrsen

DOUBLE PRECISION for pdtrsen

(global)

If *job* = 'E' or 'B', *s* is a lower bound on the reciprocal condition number for the selected cluster of eigenvalues. *s* cannot underestimate the true reciprocal condition number by more than a factor of sqrt(*n*). If *m* = 0 or *n*, *s* = 1.

If *job* = 'N' or 'V', *s* is not referenced.

sep

`REAL` for `pstrsen`

`DOUBLE PRECISION` for `pdtrsen`

(global)

If *job* = 'V' or 'B', *sep* is the estimated reciprocal condition number of the specified invariant subspace. If

m = 0 or *n*, *sep* = norm(*t*).

If *job* = 'N' or 'E', *sep* is not referenced.

work(1)

On exit, if *info* = 0, *work*(1) returns the optimal *lwork*.

iwork(1)

On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*.

info

(global) `INTEGER`

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**1000+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

> 0: here we have several possibilities

- Reordering of *t* failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); *t* may have been partially reordered, and *wr* and *wi* contain the eigenvalues in the same order as in *t*.

  On exit, *info* = {the index of *t* where the swap failed}.
- A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block.

  On exit, *info* = {the index of *t* where the swap failed}.
- If *info* = *n*+1, there is no valid BLACS context (see the BLACS documentation for details).

Application Notes

The following alignment requirements must hold:

- *mb_t* = *nb_t* = *mb_q* = *nb_q*
- *rsrc_t* = *rsrc_q*
- *csrc_t* = *csrc_q*

All matrices must be blocked by a block factor larger than or equal to two (3). This to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of *t* and *q*, i.e., *it* = *jt* = *iq* = *jq* = 1 must hold. This is however no limitation since `p?lahqr` does not compute Schur forms of submatrices anyway.

For parallel execution, use a square grid, if possible, for maximum performance. The block parameters in *para* should be kept well below the data distribution block size.

In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trti2

Computes the inverse of a triangular matrix (local unblocked algorithm).

Syntax

```
call pstrti2(uplo, diag, n, a, ia, ja, desca, info)
call pdtrti2(uplo, diag, n, a, ia, ja, desca, info)
call pctrti2(uplo, diag, n, a, ia, ja, desca, info)
call pztrti2(uplo, diag, n, a, ia, ja, desca, info)
```

Description

The p?trti2routine computes the inverse of a real/complex upper or lower triangular block matrix sub (*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

This matrix should be contained in one and only one process memory space (local operation).

Input Parameters

uplo

(global) CHARACTER*1.

Specifies whether the matrix sub (*A*) is upper or lower triangular.

= 'U': sub (*A*) is upper triangular

= 'L': sub (*A*) is lower triangular.

diag

(global) CHARACTER*1.

Specifies whether or not the matrix *A* is unit triangular.

= 'N': sub (*A*) is non-unit triangular

= 'U': sub (*A*) is unit triangular.

n

(global) INTEGER.

The number of rows and columns to be operated on, i.e., the order of the distributed submatrix sub(*A*). $n \geq 0$.

a

(local)

REAL for pstrti2

DOUBLE PRECISION for pdtrti2

COMPLEX for pctrti2

COMPLEX*16 for pztrti2.

Pointer into the local memory to an array, size (*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the triangular matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the matrix sub(*A*) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(*A*) is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the matrix sub(*A*) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(*A*) is not referenced. If *diag* = 'U', the diagonal elements of sub(*A*) are not referenced either and are assumed to be 1.

ia, *ja*

(global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a

On exit, the (triangular) inverse of the original matrix, in the same storage format.

info

INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then *info* = - (*i**100+*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?lahqr2

Updates the eigenvalues and Schur decomposition.

Syntax

```
call clahqr2 (wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
call zlahqr2 (wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
```

Description

?lahqr2 is an auxiliary routine called by ?hseqr to update the eigenvalues and Schur decomposition already computed by ?hseqr, by dealing with the Hessenberg submatrix in rows and columns *ilo* to *ihi*. This version of ?lahqr (not the standard LAPACK version) uses a double-shift algorithm (like LAPACK's ?lahqr). Unlike the standard LAPACK convention, this does not assume the subdiagonal is real, nor does it work to preserve this quality if given.

Input Parameters

wantt

LOGICAL.

= .TRUE.: the full Schur form *T* is required;

= .FALSE.: only eigenvalues are required.

wantz

LOGICAL.

= .TRUE.: the matrix of Schur vectors *Z* is required;

= .FALSE.: Schur vectors are not required.

n

INTEGER.

The order of the matrix *H*. *n* >= 0.

ilo, *ihi*

INTEGER.

It is assumed that the matrix *H* is upper triangular in rows and columns *ihi* +1 :*n*, and that matrix element *H*(*ilo*,*ilo*-1) = 0 (unless *ilo* = 1). ?lahqr works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *h* if *wantt* is .TRUE..

1 <= *ilo* <= max(1,*ihi*); *ihi* <= *n*.

h

COMPLEX for clahqr2

DOUBLE COMPLEX for zlahqr2

Array, size (*ldh*,*n*).

On entry, the upper Hessenberg matrix *H*.

ldh

INTEGER.

The leading dimension of the array *h*. *ldh* >= max(1,*n*).

iloz, *ihiz*

INTEGER.

Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE..

1 <= *iloz* <= *ilo*; *ihi* <= *ihiz* <= *n*.

z

COMPLEX for clahqr2

DOUBLE COMPLEX for zlahqr2

Array, size (*ldz*,*n*)).

If *wantz* is .TRUE., on entry *z* must contain the current matrix *Z* of transformations. If *wantz* is .FALSE., *z* is not referenced.

ldz

INTEGER.

The leading dimension of the array *z*. *ldz* >= max(1,*n*).

Output Parameters

h — On exit, if *wantt* is .TRUE., *h* is upper triangular in rows and columns *ilo*:*ihi*. If *wantt* is .FALSE., the contents of *h* are unspecified on exit.

w — COMPLEX for clahqr2

DOUBLE COMPLEX for zlahqr2

Array, size (*n*)

The computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *w*. If *wantt* is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *w*(*i*) = *h*(*i*,*i*).

z — If *wantz* is .TRUE., on exit *z* has been updated; transformations are applied only to the submatrix *Z*(*iloz*:*ihiz*,*ilo*:*ihi*). If *wantz* is .FALSE., *z* is not referenced.

info — INTEGER.

= 0: successful exit

> 0: if *info* = *i*, ?lahqr failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30*(*ihi*-*ilo*+1) iterations; elements *i*+1:*ihi* of *w* contain those eigenvalues which have been successfully computed.

?lamsh

Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.

Syntax

```
call slamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
call dlamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
call clamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
call zlamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
```

Description

The ?lamshroutine sends multiple shifts through a small (single node) matrix to see how small consecutive subdiagonal elements are modified by subsequent shifts in an effort to maximize the number of bulges that can be sent through. The subroutine should only be called when there are multiple shifts/bulges (*nbulge* > 1) and the first shift is starting in the middle of an unreduced Hessenberg matrix because of two or more small consecutive subdiagonal elements.

Input Parameters

s — (local)

REAL for slamsh

DOUBLE PRECISION for dlamsh

COMPLEX for clamsh

DOUBLE COMPLEX for zlamsh

Array of size (*lds*, 2**jblk*).

On entry, the matrix of shifts. Only the 2x2 diagonal of *s* is referenced. It is assumed that *s* has *jblk* double shifts (size 2).

lds (local) INTEGER.

On entry, the leading dimension of *S*; unchanged on exit. $1<nbulge \leq jblk \leq lds/2$.

nbulge (local) INTEGER.

On entry, the number of bulges to send through *h* (>1). *nbulge* should be less than the maximum determined (*jblk*). $1<nbulge \leq jblk \leq lds/2$.

jblk (local) INTEGER.

On entry, the number of double shifts determined for *S*; unchanged on exit.

h (local)

REAL for slamsh

DOUBLE PRECISION for dlamsh

COMPLEX for clamsh

DOUBLE COMPLEX for zlamsh

Array of size (*ldh*, *n*).

On entry, the local matrix to apply the shifts on.

h should be aligned so that the starting row is 2.

ldh (local)

INTEGER.

On entry, the leading dimension of *H*; unchanged on exit.

n (local) INTEGER.

On entry, the size of *H*. If all the bulges are expected to go through, *n* should be at least $4nbulge+2$. Otherwise, *nbulge* may be reduced by this routine.

ulp (local)

REAL for slamsh

DOUBLE PRECISION for dlamsh

REAL for clamsh

DOUBLE PRECISION for zlamsh

On entry, machine precision. Unchanged on exit.

Output Parameters

s On exit, the data is rearranged in the best order for applying.

nbulge On exit, the maximum number of bulges that can be sent through.

h	On exit, the data is destroyed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?lapst

Sorts the numbers in increasing or decreasing order.

Syntax

```
call slapst (id, n, d, indx, info )
call dlapst (id, n, d, indx, info )
```

Description

?lapst is a modified version of the LAPACK routine ?lasrt.

Define a permutation *indx* that sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D').

Use Quick Sort, reverting to Insertion sort on arrays of size <= 20. Dimension of STACK limits *n* to about 2^{32}.

Input Parameters

id	CHARACTER*1. = 'I': sort *d* in increasing order; = 'D': sort *d* in decreasing order.
n	INTEGER. The length of the array *d*.
d	REAL for slapst DOUBLE PRECISION for dlapst Array, size (*n*) The array to be sorted.

Output Parameters

indx	INTEGER. Array, size (*n*). The permutation which sorts the array *d*.
info	INTEGER. = 0: successful exit < 0: if *info* = -i, the i-th argument had an illegal value

?laqr6

Performs a single small-bulge multi-shift QR sweep collecting the transformations.

Syntax

call slaqr6(*job*, *wantt*, *wantz*, *kacc22*, *n*, *ktop*, *kbot*, *nshfts*, *sr*, *si*, *h*, *ldh*, *iloz*, *ihiz*, *z*, *ldz*, *v*, *ldv*, *u*, *ldu*, *nv*, *wv*, *ldwv*, *nh*, *wh*, *ldwh*)

call dlaqr6(*job*, *wantt*, *wantz*, *kacc22*, *n*, *ktop*, *kbot*, *nshfts*, *sr*, *si*, *h*, *ldh*, *iloz*, *ihiz*, *z*, *ldz*, *v*, *ldv*, *u*, *ldu*, *nv*, *wv*, *ldwv*, *nh*, *wh*, *ldwh*)

Description

This auxiliary subroutine called by p?laqr5 performs a single small-bulge multi-shift QR sweep, moving the chain of bulges from top to bottom in the submatrix *H*(*ktop*:*kbot*,*ktop*:*kbot*), collecting the transformations in the matrix *V or* accumulating the transformations in the matrix *Z* (see below).

This is a modified version of ?laqr5 from LAPACK 3.1.

Input Parameters

job

CHARACTER scalar

Set the kind of job to do in ?laqr6, as follows:

job = 'I': Introduce and chase bulges in submatrix

job = 'C': Chase bulges from top to bottom of submatrix

job = 'O': Chase bulges off submatrix

wantt

LOGICAL scalar

wantt= .TRUE. if the quasi-triangular Schur factor is being computed. *wantt* is set to .FALSE. otherwise.

wantz

LOGICAL scalar

wantz= .TRUE. if the orthogonal Schur factor is being computed. *wantz* is set to .FALSE. otherwise.

kacc22

INTEGER with value 0, 1, or 2.

Specifies the computation mode of far-from-diagonal orthogonal updates.

= 0: ?laqr6 does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries.

= 1: ?laqr6 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.

= 2: ?laqr6 accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.

n

INTEGER scalar

n is the order of the Hessenberg matrix *H* upon which this subroutine operates.

ktop, *kbot*

INTEGER scalar

These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either *ktop* = 1 or *H*(*ktop*,*ktop*-1) = 0 and either *kbot* = *n* or *H*(*kbot* +1,*kbot*) = 0.

nshfts INTEGER scalar

nshfts gives the number of simultaneous shifts. *nshfts* must be positive and even.

sr, *si* REAL for slaqr6

DOUBLE PRECISION for dlaqr6

Array of size *nshfts*

sr contains the real parts and *si* contains the imaginary parts of the *nshfts* shifts of origin that define the multi-shift QR sweep.

h REAL for slaqr6

DOUBLE PRECISION for dlaqr6

Array of size (*ldh*,*n*)

On input *h* contains a Hessenberg matrix .

ldh INTEGER scalar

ldh is the leading dimension of *H* just as declared in the calling procedure. *ldh*≥ max(1,*n*).

iloz, *ihiz* INTEGER scalar

Specify the rows of *z* to which transformations must be applied if *wantz*is .TRUE.. 1≤*iloz*≤*ihiz*≤*n*

z REAL for slaqr6

DOUBLE PRECISION for dlaqr6

Array of size (*ldz*,*ktop*)

If *wantz*= .TRUE., then the QR sweep orthogonal similarity transformation is accumulated into the matrix *Z*(*iloz*:*ihiz*,*kbot*:*ktop*), stored in the array *z*, from the right.

If *wantz*= .FALSE., then *z* is unreferenced.

ldz INTEGER scalar

ldz is the leading dimension of *z* just as declared in the calling procedure. *ldz*≥*n*.

v REAL for slaqr6

DOUBLE PRECISION for dlaqr6

(workspace) array of size (*ldv*,*nshfts*/2)

ldv INTEGER scalar

ldv is the leading dimension of *v* as declared in the calling procedure. *ldv*≥3.

u REAL for slaqr6

DOUBLE PRECISION for dlaqr6

(workspace) array of size (*ldu*,3*nshfts*-3)

ldu INTEGER scalar

ldu is the leading dimension of *u* just as declared in the calling subroutine. *ldu*≥3**nshfts*-3.

nh INTEGER scalar

nh is the number of columns in array *wh* available for workspace. *nh*≥1 is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.

wh REAL for slaqr6

DOUBLE PRECISION for dlaqr6

(workspace) array of size (*ldwh*,*nh*)

ldwh INTEGER scalar

Leading dimension of *wh* just as declared in the calling procedure. *ldwh*≥3**nshfts*-3.

nv INTEGER scalar

nv is the number of rows in *wv* available for workspace. *nv*≥1 is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.

wv REAL for slaqr6

DOUBLE PRECISION for dlaqr6

(workspace) array of size (*ldwv*,3**nshfts*-3)

ldwv INTEGER scalar

ldwv is the leading dimension of *wv* as declared in the in the calling subroutine. *ldwv*≥*nv*.

OUTPUT Parameters

h A multi-shift QR sweep with shifts *sr*(*j*)+*i***si*(*j*) is applied to the isolated diagonal block in rows and columns *ktop* through *kbot*.

z If *wantz* is .TRUE., then the QR sweep orthogonal/unitary similarity transformation is accumulated into the matrix *Z*(*iloz*:*ihiz*,*kbot*:*ktop*) from the right.

If *wantz* is .FALSE., then *z* is unreferenced.

Application Notes

Notes

Based on contributions by Karen Braman and Ralph Byers, Department of Mathematics, University of Kansas, USA Robert Granat, Department of Computing Science and HPC2N, Umea University, Sweden

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?lar1va

Computes scaled eigenvector corresponding to given eigenvalue.

Syntax

`call slar1va(n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )`

`call dlar1va(n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )`

Description

`?slar1va` computes the (scaled) r-th column of the inverse of the submatrix in rows `b1` through `bn` of the tridiagonal matrix $LDL^T - \lambda I$. When λ is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, `r` corresponds to the index where the eigenvector is largest in magnitude. The following steps accomplish this computation :

1. Stationary qd transform, $LDL^T - \lambda I = L_+D_+L_+^T$,
2. Progressive qd transform, $LDL^T - \lambda I = U_-D_-U_-^T$,
3. Computation of the diagonal elements of the inverse of $LDL^T - \lambda I$ by combining the above transforms, and choosing `r` as the index where the diagonal of the inverse is (one of the) largest in magnitude.
4. Computation of the (scaled) `r`-th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

`n`

`INTEGER`

The order of the matrix LDL^T.

`b1`

`INTEGER`

First index of the submatrix of LDL^T.

`bn`

`INTEGER`

Last index of the submatrix of LDL^T.

`lambda`

`REAL` for `slar1va`

`DOUBLE PRECISION` for `dlar1va`

The shift λ. In order to compute an accurate eigenvector, `lambda` should be a good approximation to an eigenvalue of LDL^T.

`l`

`REAL` for `slar1va`

`DOUBLE PRECISION` for `dlar1va`

Array of size `n`-1

The (`n`-1) subdiagonal elements of the unit bidiagonal matrix L, in elements 1 to `n`-1.

`d`

`REAL` for `slar1va`

`DOUBLE PRECISION` for `dlar1va`

Array of size `n`

The n diagonal elements of the diagonal matrix D.

ld

REAL for slar1va

DOUBLE PRECISION for dlar1va

Array of size n-1

The n-1 elements $l(i)*d(i)$.

lld

REAL for slar1va

DOUBLE PRECISION for dlar1va

Array of size n-1

The n-1 elements $l(i)*l(i)*d(i)$.

pivmin

REAL for slar1va

DOUBLE PRECISION for dlar1va

The minimum pivot in the Sturm sequence.

gaptol

REAL for slar1va

DOUBLE PRECISION for dlar1va

Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.

z

REAL for slar1va

DOUBLE PRECISION for dlar1va

Array of size n

On input, all entries of z must be set to 0.

wantnc

LOGICAL

Specifies whether *negcnt* has to be computed.

r

INTEGER

The twist index for the twisted factorization used to compute z.

On input, $0 \le r \le n$. If r is input as 0, r is set to the index where $(LDL^T - \sigma I)^{-1}$ is largest in magnitude. If $1 \le r \le n$, r is unchanged.

Ideally, r designates the position of the maximum entry in the eigenvector.

work

REAL for slar1va

DOUBLE PRECISION for dlar1va

(Workspace) array of size 4*n

OUTPUT Parameters

z

On output, z contains the (scaled) r-th column of the inverse. The scaling is such that $z(r)$ equals 1.

negcnt

INTEGER

If *wantnc*is .TRUE. then *negcnt* = the number of pivots < *pivmin* in the matrix factorization LDL^T, and *negcnt* = -1 otherwise.

ztz

REAL for slar1va

DOUBLE PRECISION for dlar1va

The square of the 2-norm of *z*.

mingma

REAL for slar1va

DOUBLE PRECISION for dlar1va

The reciprocal of the largest (in magnitude) diagonal element of the inverse of $LDL^T - \sigma I$.

r

On output, *r* contains the twist index used to compute *z*.

isuppz

INTEGER array of size 2

The support of the vector in *z*, i.e., the vector *z* is non-zero only in elements *isuppz*(1) through *isuppz*(2).

nrminv

REAL for slar1va

DOUBLE PRECISION for dlar1va

nrminv = 1/SQRT(*ztz*)

resid

REAL for slar1va

DOUBLE PRECISION for dlar1va

The residual of the FP vector.

resid = ABS(*mingma*)/SQRT(*ztz*)

rqcorr

REAL for slar1va

DOUBLE PRECISION for dlar1va

The Rayleigh Quotient correction to *lambda*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?laref

Applies Householder reflectors to matrices on their rows or columns.

Syntax

```
call slaref (type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop, itmp1,
itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3 )
```

```
call dlaref (type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop, itmp1,
itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3 )
```

```
call claref (type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop, itmp1,
itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3 )
```

```
call zlaref (type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop, itmp1,
itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3 )
```

Description

?laref applies one or several Householder reflectors of size 3 to one or two matrices (if column is specified) on either their rows or columns.

Input Parameters

type

(local)

CHARACTER*1.

If 'R': Apply reflectors to the rows of the matrix (apply from left)

Otherwise: Apply reflectors to the columns of the matrix

Unchanged on exit.

a

(local)

REAL for slaref

DOUBLE PRECISION for dlaref

COMPLEX for claref

DOUBLE COMPLEX for zlaref

Array, (*lld_a*,*LOCc*(*ja*+*n*-1))

On entry, the matrix to receive the reflections.

lda

(local)

INTEGER.

On entry, the leading dimension of *a*.

Unchanged on exit.

wantz

(local)

LOGICAL.

If .TRUE., then apply any column reflections to *z* as well.

If .FALSE., then do no additional work on *z*.

z

(local)

REAL for slaref

DOUBLE PRECISION for dlaref

COMPLEX for claref

DOUBLE COMPLEX for zlaref

Array, (*ldz*,*ncols*), where the value *ncols* depends on other arguments. If *wantz*== .TRUE. and *type*≠ 'R' then *ncols* = *icol1* + 3*(*lihiz* - *liloz* + 1). Otherwise, *ncols* is unused.

On entry, the second matrix to receive column reflections.

This is changed only if *wantz* is set.

ldz

(local)

INTEGER.

On entry, the leading dimension of *z*.

Unchanged on exit.

block (local)

LOGICAL.

If .TRUE., then apply several reflectors at once and read their data from the *vecs* array.

If .FALSE., apply the single reflector given by *v2*, *v3*, *t1*, *t2*, and *t3*.

irow1 (local)

INTEGER.

On entry, the local row element of *a*.

icol1 (local)

INTEGER.

On entry, the local column element of *a*.

istart (local)

INTEGER.

Specifies the "number" of the first reflector. This is used as an index into *vecs* if *block* is set. *istart* is ignored if *block* is .FALSE..

istop (local)

INTEGER.

Specifies the "number" of the last reflector. This is used as an index into *vecs* if *block* is set. *istop* is ignored if *block* is .FALSE..

itmp1 (local)

INTEGER.

Starting range into *a*. For rows, this is the local first column. For columns, this is the local first row.

itmp2 (local)

INTEGER.

Ending range into *a*. For rows, this is the local last column. For columns, this is the local last row.

liloz, *lihiz* (local)

INTEGER.

These serve the same purpose as *itmp1*, *itmp2* but for *z* when *wantz* is set.

vecs (local)

`REAL` for `slaref`

`DOUBLE PRECISION` for `dlaref`

`COMPLEX` for `claref`

	`DOUBLE COMPLEX` for `zlaref` Array of size 3*N (matrix size) This holds the size 3 reflectors one after another and this is only accessed when *block* is .TRUE.

v2, *v3*, *t1*, *t2*, *t3*	(local)
	`REAL` for `slaref`
	`DOUBLE PRECISION` for `dlaref`
	`COMPLEX` for `claref`
	`DOUBLE COMPLEX` for `zlaref`
	This holds information on a single size 3 Householder reflector and is read when *block* is .FALSE., and overwritten when *block* is .TRUE.

Output Parameters

a	The updated matrix on exit.
z	This is changed only if *wantz* is set.
irow1	Undefined on output.
icol1	Undefined on output.
v2, *v3*, *t1*, *t2*, *t3*	Overwritten when *block* is .TRUE.

?larrb2

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

```
call slarrb2( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,
iwork, pivmin, lgpvmn, lgspdm, twist, info )
```

```
call dlarrb2( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,
iwork, pivmin, lgpvmn, lgspdm, twist, info )
```

Description

Given the relatively robust representation (RRR) LDL^T, `?larrb2` does "limited" bisection to refine the eigenvalues of LDL^T, *w*(*ifirst* - *offset*) through *w*(*ilast* - *offset*), to more accuracy. Initial guesses for these eigenvalues are input in *w*, the corresponding estimate of the error in these guesses and their gaps are input in *werr* and *wgap*, respectively. During bisection, intervals [*left*, *right*] are maintained by storing their mid-points and semi-widths in the arrays *w* and *werr* respectively.

NOTE
There are very few minor differences between larrb from LAPACK and this current subroutine ?larrb2. The most important reason for creating this nearly identical copy is profiling: in the ScaLAPACK MRRR algorithm, eigenvalue computation using ?larrb2 is used for refinement in the construction of the representation tree, as opposed to the initial computation of the eigenvalues for the root RRR which uses ?larrb. When profiling, this allows an easy quantification of refinement work vs. computing eigenvalues of the root.

Input Parameters

n INTEGER

The order of the matrix.

d REAL for slarrb2

DOUBLE PRECISION for dlarrb2

Array of size *n*.

The *n* diagonal elements of the diagonal matrix *D*.

lld REAL for slarrb2

DOUBLE PRECISION for dlarrb2

Array of size *n*-1.

The (*n*-1) elements $l_i * l_i * d(i)$.

ifirst INTEGER

The index of the first eigenvalue to be computed.

ilast INTEGER

The index of the last eigenvalue to be computed.

rtol1, *rtol2* REAL for slarrb2

DOUBLE PRECISION for dlarrb2

Tolerance for the convergence of the bisection intervals.

An interval [*left*, *right*] has converged if *right* - *left* < max (*rtol1* * *gap*, *rtol2* * max(|*left*|, |*right*|)) where *gap* is the (estimated) distance to the nearest eigenvalue.

offset INTEGER

Offset for the arrays *w*, *wgap* and *werr*, i.e., the *ifirst* - *offset* through *ilast* - *offset* elements of these arrays are to be used.

w REAL for slarrb2

DOUBLE PRECISION for dlarrb2

Array of size *n*

On input, *w*(*ifirst* - *offset*) through *w*(*ilast*- *offset*) are estimates of the eigenvalues of LDL^T indexed *ifirst* through *ilast*.

wgap REAL for slarrb2

DOUBLE PRECISION for dlarrb2

Array of size *n*-1.

On input, the (estimated) gaps between consecutive eigenvalues of LDL^T, i.e., *wgap*(*I* - *offset*) is the gap between eigenvalues *I* and *I* + 1. Note that if *ifirst* = *ilast* then *wgap*(*ifirst* - *offset*) must be set to zero.

werr

REAL for slarrb2

DOUBLE PRECISION for dlarrb2

Array of size *n*.

On input, *werr*(*ifirst* - *offset*) through *werr*(*ilast* - *offset*) are the errors in the estimates of the corresponding elements in *w*.

work

REAL for slarrb2

DOUBLE PRECISION for dlarrb2

(workspace) array of size 4**n*.

Workspace.

iwork

(workspace) INTEGER array of size 2**n*.

Workspace.

pivmin

REAL for slarrb2

DOUBLE PRECISION for dlarrb2

The minimum pivot in the Sturm sequence.

lgpvmn

REAL for slarrb2

DOUBLE PRECISION for dlarrb2

Logarithm of *pivmin*, precomputed.

lgspdm

REAL for slarrb2

DOUBLE PRECISION for dlarrb2

Logarithm of the spectral diameter, precomputed.

twist

INTEGER

The twist index for the twisted factorization that is used for the *negcount*.

twist = *n*: Compute *negcount* from $LDL^T - \lambda I = L_+ D_+ L_+^T$

twist = 1: Compute *negcount* from $LDL^T - \lambda I = U_- D_- U_-^T$

twist = *r*, 1 < *r* < *n*: Compute *negcount* from $LDL^T - \lambda I = N_r \Delta_r N_r^T$

OUTPUT Parameters

w

On output, the eigenvalue estimates in *w* are refined.

wgap

On output, the eigenvalue gaps in *wgap* are refined.

werr

On output, the errors in *werr* are refined.

info

INTEGER

Error flag.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?larrd2

Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Syntax

```
call slarrd2( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplit, isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, dol, dou, info )
```

```
call dlarrd2( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplit, isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, dol, dou, info )
```

Description

`?larrd2` computes the eigenvalues of a symmetric tridiagonal matrix *T* to limited initial accuracy. This is an auxiliary code to be called from larre2a.

`?larrd2` has been created using the LAPACK code larrd which itself stems from stebz. The motivation for creating `?larrd2` is efficiency: When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, `?larrd2` can skip over blocks which contain none of the eigenvalues from DOL to DOU for which the processor responsible. In extreme cases (such as large matrices consisting of many blocks of small size like 2x2), the gain can be substantial.

Input Parameters

range

`CHARACTER`

= 'A': ("All") all eigenvalues will be found.

= 'V': ("Value") all eigenvalues in the half-open interval (*vl*, *vu*] will be found.

= 'I': ("Index") the *il*-th through *iu*-th eigenvalues (of the entire matrix) will be found.

order

`CHARACTER`

= 'B': ("By Block") the eigenvalues will be grouped by split-off block (see *iblock*, *isplit*) and ordered from smallest to largest within the block.

= 'E': ("Entire matrix") the eigenvalues for the entire matrix will be ordered from smallest to largest.

n

`INTEGER`

The order of the tridiagonal matrix *T*. *n* >= 0.

vl, *vu*

`REAL` for `slarrd2`

DOUBLE PRECISION for dlarrd2

If *range*='V', the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to *vl*, or greater than *vu*, will not be returned. *vl* < *vu*.

Not referenced if *range* = 'A' or 'I'.

il, *iu*

INTEGER

If *range*='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned.

1 ≤*il*≤*iu*≤=*n*, if *n* > 0; *il* = 1 and *iu* = 0 if *n* = 0.

Not referenced if *range* = 'A' or 'V'.

gers

REAL for slarrd2

DOUBLE PRECISION for dlarrd2

Array of size 2**n*

The *n* Gerschgorin intervals (the *i*-th Gerschgorin interval is (*gers*(2**i*-1), *gers*(2**i*))).

reltol

REAL for slarrd2

DOUBLE PRECISION for dlarrd2

The minimum relative width of an interval. When an interval is narrower than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least radix*machine epsilon.

d

REAL for slarrd2

DOUBLE PRECISION for dlarrd2

Array of size *n*

The *n* diagonal elements of the tridiagonal matrix *T*.

e

REAL for slarrd2

DOUBLE PRECISION for dlarrd2

Array of size *n*-1

The (*n*-1) off-diagonal elements of the tridiagonal matrix *T*.

e2

REAL for slarrd2

DOUBLE PRECISION for dlarrd2

Array of size *n*-1

The (*n*-1) squared off-diagonal elements of the tridiagonal matrix *T*.

pivmin

REAL for slarrd2

DOUBLE PRECISION for dlarrd2

The minimum pivot allowed in the sturm sequence for *T*.

nsplit

INTEGER

The number of diagonal blocks in the matrix *T*.

1 ≤*nsplit*≤*n*.

isplit INTEGERArray of size *n*

The splitting points, at which *T* breaks up into submatrices.

The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th submatrix consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.

(Only the first *nsplit* elements will actually be used, but since the user cannot know a priori what value *nsplit* will have, *n* words must be reserved for *isplit*.)

work REAL for slarrd2

DOUBLE PRECISION for dlarrd2

(workspace) Array of size 4**n*

iwork (workspace) INTEGER Array of size 3**n*

dol, *dou* INTEGER

Specifying an index range *dol*:*dou* allows the user to work on only a selected part of the representation tree.

Otherwise, the setting *dol*=1, *dou*=*n* should be applied.

Note that *dol* and *dou* refer to the order in which the eigenvalues are stored in W.

OUTPUT Parameters

m INTEGER

The actual number of eigenvalues found. 0 ≤*m*≤*n*.

(See also the description of *info*=2,3.)

w REAL for slarrd2

DOUBLE PRECISION for dlarrd2

Array of size *n*

On exit, the first *m* elements of *w* will contain the eigenvalue approximations. ?larrd2 computes an interval $I_j = (a_j, b_j]$ that includes eigenvalue *j*. The eigenvalue approximation is given as the interval midpoint *w*(*j*)= $(a_j + b_j)/2$. The corresponding error is bounded by *werr*(*j*) = abs(a_j - b_j)/2.

werr REAL for slarrd2

DOUBLE PRECISION for dlarrd2

Array of size *n*

The error bound on the corresponding eigenvalue approximation in *w*.

wl, *wu* REAL for slarrd2

DOUBLE PRECISION for dlarrd2

The interval (*wl*, *wu*] contains all the wanted eigenvalues.

If *range*='V', then *wl*=*vl* and *wu*=*vu*.

If *range*='A', then *wl* and *wu* are the global Gerschgorin bounds

on the spectrum.

If *range*='I', then *wl* and *wu* are computed by SLAEBZ from the

index range specified.

iblock

INTEGERArray of size *n*

At each row/column *j* where *e*(*j*) is zero or small, the matrix *T* is considered to split into a block diagonal matrix. On exit, if *info* = 0, *iblock*(*i*) specifies to which block (from 1 to the number of blocks) the eigenvalue *w*(*i*) belongs. (?larrd2 may use the remaining *n*-*m* elements as workspace.)

indexw

INTEGERArray of size *n*

The indices of the eigenvalues within each block (submatrix); for example, *indexw*(*i*)= *j* and *iblock*(*i*)=*k* imply that the *i*-th eigenvalue *w*(*i*) is the *j*-th eigenvalue in block *k*.

info

INTEGER

= 0: successful exit

< 0: if *info* = -*i*, the *i*-th argument had an illegal value

> 0: some or all of the eigenvalues failed to converge or were not computed:

- =1 or 3: Bisection failed to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.
- =2 or 3: *range*='I' only: Not all of the eigenvalues *il*:*iu* were found.
- = 4: *range*='I', and the Gershgorin interval initially used was too small. No eigenvalues were computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?larre2

Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

call slarre2(*range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit, m, dol, dou, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info*)

call dlarre2(*range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit, m, dol, dou, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info*)

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix *T*, `?larre2` sets, via `?larra`, "small" off-diagonal elements to zero. For each block T_i, it finds

- a suitable shift at one end of the block's spectrum,
- the root RRR, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

The representations and eigenvalues found are then returned to `?stegr2` to compute the eigenvectors *T*.

`?larre2` is more suitable for parallel computation than the original LAPACK code for computing the root RRR and its eigenvalues. When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, `?larre2` can skip over blocks which contain none of the eigenvalues from *dol* to *dou* for which the processor is responsible. In extreme cases (such as large matrices consisting of many blocks of small size, e.g. 2x2), the gain can be substantial.

Input Parameters

range

`CHARACTER`

= 'A': ("All") all eigenvalues will be found.

= 'V': ("Value") all eigenvalues in the half-open interval (*vl*, *vu*] will be found.

= 'I': ("Index") the *il*-th through *iu*-th eigenvalues (of the entire matrix) will be found.

n

`INTEGER`

The order of the matrix. $n > 0$.

vl, *vu*

`REAL` for `slarre2`

`DOUBLE PRECISION` for `dlarre2`

If *range*='V', the lower and upper bounds for the eigenvalues.

Eigenvalues less than or equal to *vl*, or greater than *vu*, will not be returned. $vl < vu$.

il, *iu*

`INTEGER`

If *range*='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned.

$1 \le il \le iu \le n$.

d

`REAL` for `slarre2`

`DOUBLE PRECISION` for `dlarre2`

Array of size *n*

The *n* diagonal elements of the tridiagonal matrix *T*.

e

REAL for slarre2

DOUBLE PRECISION for dlarre2

Array of size *n*

The first (*n*-1) entries contain the subdiagonal elements of the tridiagonal matrix *T*; *e*(*n*) need not be set.

e2

REAL for slarre2

DOUBLE PRECISION for dlarre2

Array of size *n*

The first (*n*-1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix *T*; *e2*(*n*) need not be set.

rtol1, *rtol2*

REAL for slarre2

DOUBLE PRECISION for dlarre2

Parameters for bisection.

An interval [*left*, *right*] has converged if *right-left*<max(*rtol1***gap*, *rtol2**max(|*left*|,|*right*|))

spltol

REAL for slarre2

DOUBLE PRECISION for dlarre2

The threshold for splitting.

dol, *dou*

INTEGER

Specifying an index range *dol*:*dou* allows the user to work on only a selected part of the representation tree. Otherwise, the setting *dol*=1, *dou*=*n* should be applied.

Note that *dol* and *dou* refer to the order in which the eigenvalues are stored in *w*.

work

REAL for slarre2

DOUBLE PRECISION for dlarre2

Workspace array of size 6**n*

iwork

INTEGER

Workspace array of size 5**n*

OUTPUT Parameters

vl, *vu*

If *range*='I' or ='A', ?larre2 contains bounds on the desired part of the spectrum.

d

The *n* diagonal elements of the diagonal matrices D_i.

e

e contains the subdiagonal elements of the unit bidiagonal matrices L_i. The entries *e*(*isplit*(*i*)), 1 ≤*j*≤*nsplit*, contain the base points σ_i on output.

e2

The entries *e2*(*isplit*(*i*)), 1 ≤*j*≤*nsplit*, are set to zero.

nsplit INTEGER

The number of blocks *T* splits into. 1 ≤*nsplit*≤*n*.

isplit INTEGERArray of size *n*

The splitting points, at which *T* breaks up into blocks.

The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th block consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.

m INTEGER

The total number of eigenvalues (of all $L_iD_iL_i^T$) found.

w REAL for slarre2

DOUBLE PRECISION for dlarre2

Array of size *n*

The first *m* elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_iD_iL_i^T$, are sorted in ascending order (?larre2 may use the remaining *n*-*m* elements as workspace).

Note that immediately after exiting this routine, only the eigenvalues from position *dol*:*dou* in *w* might rely on this processor when the eigenvalue computation is done in parallel.

werr REAL for slarre2

DOUBLE PRECISION for dlarre2

Array of size *n*

The error bound on the corresponding eigenvalue in *w*.

Note that immediately after exiting this routine, only the uncertainties from position *dol*:*dou* in *werr* might rely on this processor when the eigenvalue computation is done in parallel.

wgap REAL for slarre2

DOUBLE PRECISION for dlarre2

Array of size *n*

The separation from the right neighbor eigenvalue in *w*.

The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree.

Exception: at the right end of a block we store the left gap

Note that immediately after exiting this routine, only the gaps from position *dol*:*dou* in *wgap* might rely on this processor when the eigenvalue computation is done in parallel.

iblock INTEGERArray of size *n*

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in *w*; *iblock*(*i*)=1 if eigenvalue *w*(*i*) belongs to the first block from the top, *iblock*(*i*)=2 if *w*(*i*) belongs to the second block, and so on.

indexw

INTEGERArray of size *n*

The indices of the eigenvalues within each block (submatrix); for example, *indexw*(*i*)= 10 and *iblock*(*i*)=2 imply that the *i*-th eigenvalue *w*(*i*) is the 10th eigenvalue in block 2.

gers

REAL for slarre2

DOUBLE PRECISION for dlarre2

Array of size 2**n*

The *n* Gerschgorin intervals (the *i*-th Gerschgorin interval is (*gers*(2**i*-1), *gers*(2**i*))).

pivmin

REAL for slarre2

DOUBLE PRECISION for dlarre2

The minimum pivot in the sturm sequence for *T*.

info

INTEGER

= 0: successful exit

> 0: A problem occurred in ?larre2.

< 0: One of the called subroutines signaled an internal problem.

Needs inspection of the corresponding parameter *info* for further information.

=-1: Problem in ?larrd.

=-2: Not enough internal iterations to find the base representation.

=-3: Problem in ?larrb when computing the refined root representation for ?lasq2.

=-4: Problem in ?larrb when preforming bisection on the desired part of the spectrum.

=-5: Problem in ?lasq2

=-6: Problem in ?lasq2

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?larre2a

Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

```
call slarre2a( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit,
m, dol, dou, needil, neediu, w, werr, wgap, iblock, indexw, gers, sdiam, pivmin, work,
iwork, minrgp, info )
```

```
call dlarre2a( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit,
m, dol, dou, needil, neediu, w, werr, wgap, iblock, indexw, gers, sdiam, pivmin, work,
iwork, minrgp, info )
```

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T, `?larre2a` sets any "small" off-diagonal elements to zero, and for each unreduced block T_i, it finds

- a suitable shift at one end of the block's spectrum,
- the base representation, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

> **NOTE**
> The algorithm obtains a crude picture of all the wanted eigenvalues (as selected by *range*). However, to reduce work and improve scalability, only the eigenvalues *dol* to *dou* are refined. Furthermore, if the matrix splits into blocks, RRRs for blocks that do not contain eigenvalues from *dol* to *dou* are skipped. The DQDS algorithm (subroutine `?lasq2`) is not used, unlike in the sequential case. Instead, eigenvalues are computed in parallel to some figures using bisection.

Input Parameters

range

`CHARACTER`

= 'A': ("All") all eigenvalues will be found.

= 'V': ("Value") all eigenvalues in the half-open interval (*vl*, *vu*] will be found.

= 'I': ("Index") the *il*-th through *iu*-th eigenvalues (of the entire matrix) will be found.

n

`INTEGER`

The order of the matrix. $n > 0$.

vl, *vu*

`REAL` for `slarre2a`

`DOUBLE PRECISION` for `dlarre2a`

If *range*='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to *vl*, or greater than *vu*, will not be returned. $vl < vu$.

If *range*='I' or ='A', `?larre2a` computes bounds on the desired part of the spectrum.

il, *iu*

`INTEGER`

If *range*='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned.

$1 \le il \le iu \le n$.

d

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Array of size *n*

On entry, the *n* diagonal elements of the tridiagonal matrix *T*.

e

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Array of size *n*

The first (*n*-1) entries contain the subdiagonal elements of the tridiagonal matrix *T*; *e*(*n*) need not be set.

e2

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Array of size *n*

The first (*n*-1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix *T*; *e2*(*n*) need not be set.

rtol1, *rtol2*

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Parameters for bisection.

An interval [*left*,*right*] has converged if *right* - *left* < max(*rtol1***gap*, *rtol2**max(|*left*|,|*right*|))

spltol

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

The threshold for splitting.

dol, *dou*

INTEGER

If the user wants to work on only a selected part of the representation tree, he can specify an index range *dol*:*dou*.

Otherwise, the setting *dol*=1, *dou*=*n* should be applied.

Note that *dol* and *dou* refer to the order in which the eigenvalues are stored in *w*.

work

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Workspace array of size 6**n*

iwork

INTEGER

Workspace array of size 5**n*

minrgp

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.

OUTPUT Parameters

vl, *vu*

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

If *range*='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to *vl*, or greater than *vu*, are not returned. *vl* < *vu*.

If *range*='I' or *range*='A', ?larre2a computes bounds on the desired part of the spectrum.

d

The *n* diagonal elements of the diagonal matrices D_i.

e

e contains the subdiagonal elements of the unit bidiagonal matrices L_i. The entries *e*(*isplit*(*i*)), 1 ≤*i*≤*nsplit*, contain the base points σ_i on output.

e2

The entries *e2*(*isplit*(*i*)), 1 ≤*i*≤*nsplit* have been set to zero.

nsplit

INTEGER

The number of blocks *T* splits into. 1 ≤*nsplit*≤*n*.

isplit

INTEGER

Array of size *n*

The splitting points, at which *T* breaks up into blocks.

The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th block consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.

m

INTEGER

The total number of eigenvalues (of all $L_iD_iL_i^T$) found.

needil, *neediu*

INTEGER

The indices of the leftmost and rightmost eigenvalues of the root node RRR which are needed to accurately compute the relevant part of the representation tree.

w

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Array of size *n*

The first *m* elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_iD_iL_i^T$, are sorted in ascending order (?larre2a may use the remaining *n*-*m* elements as workspace).

Note that immediately after exiting this routine, only the eigenvalues from position *dol*:*dou* in *w* rely on this processor because the eigenvalue computation is done in parallel.

werr

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Array of size *n*

The error bound on the corresponding eigenvalue in *w*.

Note that immediately after exiting this routine, only the uncertainties from position *dol*:*dou* in *werr* are reliable on this processor because the eigenvalue computation is done in parallel.

wgap

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Array of size *n*

The separation from the right neighbor eigenvalue in *w*. The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree.

Exception: at the right end of a block we store the left gap

Note that immediately after exiting this routine, only the gaps from position *dol*:*dou* in *wgap* are reliable on this processor because the eigenvalue computation is done in parallel.

iblock

INTEGERArray of size *n*

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in *w*; *iblock*(*i*)=1 if eigenvalue *w*(*i*) belongs to the first block from the top, *iblock*(*i*)=2 if *w*(*i*) belongs to the second block, and so on.

indexw

INTEGERArray of size *n*

The indices of the eigenvalues within each block (submatrix); for example, *indexw*(*i*)= 10 and *iblock*(*i*)=2 imply that the *i*-th eigenvalue *w*(*i*) is the 10th eigenvalue in block 2.

gers

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Array of size 2*n*

The *n* Gerschgorin intervals (the *i*-th Gerschgorin interval is (*gers*(2**i*-1), *gers*(2**i*))).

pivmin

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

The minimum pivot in the sturm sequence for *T*.

info

INTEGER

= 0: successful exit

> 0: A problem occurred in ?larre2a.

< 0: One of the called subroutines signaled an internal problem. Needs inspection of the corresponding parameter *info* for further information.

=-1: Problem in ?larrd2.

=-2: Not enough internal iterations to find base representation.

=-3: Problem in ?larrb2 when computing the refined root representation.

=-4: Problem in ?larrb2 when preforming bisection on the desired part of the spectrum.

= -9 Problem: *m* < *dou-dol*+1, that is the code found fewer eigenvalues than it was supposed to.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?larrf2

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

```
call slarrf2( n, d, l, ld, clstrt, clend, clmid1, clmid2, w, wgap, werr, trymid, spdiam, clgapl, clgapr, pivmin, sigma, dplus, lplus, work, info )
```

```
call dlarrf2( n, d, l, ld, clstrt, clend, clmid1, clmid2, w, wgap, werr, trymid, spdiam, clgapl, clgapr, pivmin, sigma, dplus, lplus, work, info )
```

Description

Given the initial representation LDL^T and its cluster of close eigenvalues (in a relative measure), *w*(*clstrt*), *w*(*clstrt*+1), ... *w*(*clend*), ?larrf2 finds a new relatively robust representation $LDL^T - \sigma I = L_+D_+L_+^T$ such that at least one of the eigenvalues of $L_+D_+L_+^T$ is relatively isolated.

This is an enhanced version of ?larrf that also tries shifts in the middle of the cluster, should there be a large gap, in order to break large clusters into at least two pieces.

Input Parameters

n

INTEGER

The order of the matrix (subblock, if the matrix was split).

d

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array of size *n*

The *n* diagonal elements of the diagonal matrix *D*.

l

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array of size *n*-1

The (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L*.

ld

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array of size *n*-1

The (*n*-1) elements *l*(*i*)**d*(*i*).

clstrt

INTEGER

The index of the first eigenvalue in the cluster.

clend INTEGER

The index of the last eigenvalue in the cluster.

clmid1, *clmid2* INTEGER

The index of a middle eigenvalue pair with large gap.

w REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array of size ≥ (*clend*-*clstrt*+1)

The eigenvalue approximations of *LD* L^T in ascending order. *w*(*clstrt*) through *w*(*clend*) form the cluster of relatively close eigenalues.

wgap REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array of size ≥ (*clend*-*clstrt*+1)

The separation from the right neighbor eigenvalue in *w*.

werr REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array of size ≥ (*clend*-*clstrt*+1)

werr contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in *w*.

spdiam REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Estimate of the spectral diameter obtained from the Gerschgorin intervals

clgapl, *clgapr* REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Absolute gap on each end of the cluster.

Set by the calling routine to protect against shifts too close to eigenvalues outside the cluster.

pivmin REAL for slarrf2

DOUBLE PRECISION for dlarrf2

The minimum pivot allowed in the Sturm sequence.

work REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Workspace array of size 2**n*

OUTPUT Parameters

wgap Contains refined values of its input approximations. Very small gaps are unchanged.

sigma
REAL for slarrf2
DOUBLE PRECISION for dlarrf2
The shift (σ) used to form $L_+D_+L_+^T$.

dplus
REAL for slarrf2
DOUBLE PRECISION for dlarrf2
Array of size *n*
The *n* diagonal elements of the diagonal matrix D_+.

lplus
REAL for slarrf2
DOUBLE PRECISION for dlarrf2
Array of size *n*-1
The first (*n*-1) elements of *lplus* contain the subdiagonal elements of the unit bidiagonal matrix L_+.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?larrv2

*Computes the eigenvectors of the tridiagonal matrix T = L*D*L^T given L, D and the eigenvalues of L*D*L^T.*

Syntax

```
call slarrv2( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, needil, neediu, minrgp,
rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, sdiam, z, ldz, isuppz, work, iwork,
vstart, finish, maxcls, ndepth, parity, zoffset, info )
```

```
call dlarrv2( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, needil, neediu, minrgp,
rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, sdiam, z, ldz, isuppz, work, iwork,
vstart, finish, maxcls, ndepth, parity, zoffset, info )
```

Description

?larrv2 computes the eigenvectors of the tridiagonal matrix $T = LDL^T$ given L, D and approximations to the eigenvalues of LDL^T. The input eigenvalues should have been computed by larre2a or by previous calls to ?larrv2.

The major difference between the parallel and the sequential construction of the representation tree is that in the parallel case, not all eigenvalues of a given cluster might be computed locally. Other processors might "own" and refine part of an eigenvalue cluster. This is crucial for scalability. Thus there might be communication necessary before the current level of the representation tree can be parsed.

Please note:

- The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy *m*≥*dou*≥*dol*≥1. These parameters are only relevant when both eigenvalues and eigenvectors are computed (stegr2b parameter *jobz* = 'V'). ?larrv2 only computes the eigenvectors corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenvectors belonging to *w*(1) through *w*(*m*), only the eigenvectors belonging to eigenvalues *w*(*dol*) through *w*(*dou*) are computed. In this case, only the eigenvalues *dol*:*dou* are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.

- The additional arguments *vstart*, *finish*, *ndepth*, *parity*, *zoffset* are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable *finish* is true, the computation has ended. All eigenpairs between *dol* and *dou* have been computed. *m* is set to *dou* - *dol* + 1.
- ?larrv2 needs more workspace in *z* than the sequential slarrv. It is used to store the conformal embedding of the local representation tree.

Input Parameters

n

INTEGER

The order of the matrix. $n \geq 0$.

vl, *vu*

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Lower and upper bounds of the interval that contains the desired eigenvalues. $vl < vu$. Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range. *vu* is currently not used but kept as parameter in case needed.

d

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size *n*

The *n* diagonal elements of the diagonal matrix *d*. On exit, *d* is overwritten.

l

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size *n*

The (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L* are in elements 1 to *n*-1 of *l* (if the matrix is not split.) At the end of each block is stored the corresponding shift as given by ?larre. On exit, *l* is overwritten.

pivmin

DOUBLE PRECISION

The minimum pivot allowed in the sturm sequence.

isplit

INTEGER

Array of size *n*

The splitting points, at which *T* breaks up into blocks. The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1) + 1 through *isplit*(2), etc.

m INTEGER

The total number of input eigenvalues. 0 ≤*m*≤*n*.

dol, *dou* INTEGER

If you want to compute only selected eigenvectors from all the eigenvalues supplied, you can specify an index range *dol*:*dou*. Or else the setting *dol*=1, *dou*=*m* should be applied. Note that *dol* and *dou* refer to the order in which the eigenvalues are stored in *w*. If you want to compute only selected eigenpairs, the columns *dol*-1 to *dou*+1 of the eigenvector space *z* contain the computed eigenvectors. All other columns of *z* are set to zero.

If *dol* > 1, then *z*(:,*dol*-1-*zoffset*) is used.

If *dou* < *m*, then *z*(:,*dou*+1-*zoffset*) is used.

needil, *neediu* INTEGER

Describe which are the left and right outermost eigenvalues that still need to be included in the computation. These indices indicate whether eigenvalues from other processors are needed to correctly compute the conformally embedded representation tree.

When *dol*≤*needil*≤*neediu*≤*dou*, all required eigenvalues are local to the processor and no communication is required to compute its part of the representation tree.

minrgp REAL for slarrv2

DOUBLE PRECISION for dlarrv2

The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.

rtol1, *rtol2* REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Parameters for bisection. An interval [*left*,*right*] has converged if *right*-*left* < max(*rtol1***gap*, *rtol2**max(|*left*|,|*right*|))

w REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size *n*

The first *m* elements of *w* contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array *w* from ?stegr2a is expected here.) Furthermore, they are with respect to the shift of the corresponding root representation for their block.

werr REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size *n*

The first *m* elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in *w*.

wgap

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size *n*

The separation from the right neighbor eigenvalue in *w*.

iblock

INTEGERArray of size *n*

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in *w*; *iblock*(*i*)=1 if eigenvalue *w*(*i*) belongs to the first block from the top, *iblock*(*i*)=2 if *w*(*i*) belongs to the second block, and so on.

indexw

INTEGERArray of size *n*

The indices of the eigenvalues within each block (submatrix). For example: *indexw*(*i*)= 10 and *iblock*(*i*)=2 imply that the *i*-th eigenvalue *w*(*i*) is the 10th eigenvalue in block 2.

gers

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size 2**n*

The *n* Gerschgorin intervals (the *i*-th Gerschgorin interval is (*gers*(2**i*-1), *gers*(2**i*))). The Gerschgorin intervals should be computed from the original unshifted matrix.

Not used but kept as parameter for possible future use.

sdiam

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size *n*

The spectral diameters for all unreduced blocks.

ldz

INTEGER

The leading dimension of the array *z*. *ldz*≥ 1, and if stegr2b parameter *jobz* = 'V', *ldz*≥ max(1,*n*).

work

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

(workspace) array of size 12**n*

iwork

(workspace) INTEGERArray of size 7**n*

vstart

LOGICAL

.TRUE. on initialization, set to .FALSE. afterwards.

finish

LOGICAL

A flag that indicates whether all eigenpairs have been computed.

maxcls

INTEGER

The largest cluster worked on by this processor in the representation tree.

ndepth

INTEGER

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

parity INTEGER

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

zoffset INTEGER

Offset for storing the eigenpairs when *z* is distributed in 1D-cyclic fashion.

OUTPUT Parameters

needil, *neediu*

w Unshifted eigenvalues for which eigenvectors have already been computed.

werr Contains refined values of its input approximations.

wgap Contains refined values of its input approximations. Very small gaps are changed.

z REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array, dimension (*ldz*, max(1,*m*))

If *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the input eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

In the distributed version, only a subset of columns is accessed, see *dol*, *dou*, and *zoffset*.

isuppz INTEGER

Array of size 2*max(1,*m*)

The support of the eigenvectors in *z*, i.e., the indices indicating the non-zero elements in *z*. The *i*-th eigenvector is non-zero only in elements *isuppz*(2**i*-1) through *isuppz*(2**i*).

vstart .TRUE. on initialization, set to .FALSE. afterwards.

finish A flag that indicates whether all eigenpairs have been computed.

maxcls The largest cluster worked on by this processor in the representation tree.

ndepth The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

parity An internal parameter needed for the storage of the clusters on the current level of the representation tree.

info INTEGER

= 0: successful exit

> 0: A problem occured in ?larrv2.

< 0: One of the called subroutines signaled an internal problem.

Needs inspection of the corresponding parameter *info* for further information.

=-1: Problem in ?larrb2 when refining a child's eigenvalues.

=-2: Problem in ?larrf2 when computing the RRR of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter *minrgp* smaller and recompile. However, as the orthogonality of the computed vectors is proportional to 1/*minrgp*, be aware that decreasing *minrgp* might be reduce precision.

=-3: Problem in ?larrb2 when refining a single eigenvalue after the Rayleigh correction was rejected.

= 5: The Rayleigh Quotient Iteration failed to converge to full accuracy.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?lasorte

Sorts eigenpairs by real and complex data types.

Syntax

```
call slasorte(s, lds, j, out, info)
call dlasorte(s, lds, j, out, info)
```

Description

The ?lasorteroutine sorts eigenpairs so that real eigenpairs are together and complex eigenpairs are together. This helps to employ 2x2 shifts easily since every second subdiagonal is guaranteed to be zero. This routine does no parallel work and makes no calls.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

s (local)

REAL for slasorte

DOUBLE PRECISION for dlasorte

Array of size *lds*.

On entry, a matrix already in Schur form.

lds (local) INTEGER.

On entry, the leading dimension of the array *s*; unchanged on exit.

j (local) INTEGER.

On entry, the order of the matrix *S*; unchanged on exit.

out	(local) REAL for slasorte DOUBLE PRECISION for dlasorte Array of size 2**j*. The work buffer required by the routine.
info	(local) INTEGER. Set, if the input matrix had an odd number of real eigenvalues and things could not be paired or if the input matrix *S* was not originally in Schur form. 0 indicates successful completion.

Output Parameters

s	On exit, the diagonal blocks of *S* have been rewritten to pair the eigenvalues. The resulting matrix is no longer similar to the input.
out	Work buffer.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?lasrt2

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt2(id, n, d, key, info)
call dlasrt2(id, n, d, key, info)
```

Description

The ?lasrt2routine is modified LAPACK routine ?lasrt, which sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20. The size of STACK limits *n* to about 2^{32}.

Input Parameters

id	CHARACTER*1. = 'I': sort *d* in increasing order; = 'D': sort *d* in decreasing order.
n	INTEGER. The length of the array *d*.
d	REAL for slasrt2 DOUBLE PRECISION for dlasrt2. Array of size *n*. On entry, the array to be sorted.
key	INTEGER. Array of size *n*. On entry, *key* contains a key to each of the entries in d().

Typically, *key*(*i*) = *i* for all *i*.

Output Parameters

d

On exit, *d* has been sorted into increasing order

(*d*(1) ≤ ... ≤ *d*(*n*))

or into decreasing order

(*d*(1) ≥ ... ≥ *d*(*n*)),

depending on *id*.

info

INTEGER.

= 0: successful exit

< 0: if *info* = -i, the *i*-th argument had an illegal value.

key

On exit, *key* is permuted in exactly the same manner as *d* was permuted from input to output. Therefore, if *key*(*i*) = *i* for all *i* on input, *d*(*i*) on output equals *d*(*key*(*i*)) on input.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?stegr2

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstegr2( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, work, lwork, iwork, liwork, dol, dou, zoffset, info )
```

```
call dstegr2( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, work, lwork, iwork, liwork, dol, dou, zoffset, info )
```

Description

?stegr2 computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*. It is invoked in the ScaLAPACK MRRR driver p?syevr and the corresponding Hermitian version either when only eigenvalues are to be computed, or when only a single processor is used (the sequential-like case).

?stegr2 has been adapted from LAPACK's ?stegr. Please note the following crucial changes.

1. The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy *m*≥*dou*≥*dol*≥1. ?stegr2*only* computes the eigenpairs corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenpairs belonging to *w*(1) through *w*(*m*), only the eigenvectors belonging to eigenvalues *w*(*dol*) through *w*(*dou*) are computed. In this case, only the eigenvalues *dol*:*dou* are guaranteed to be fully accurate.
2. *m* is *not* the number of eigenvalues specified by *range*, but is *m* = *dou* - *dol* + 1. This concerns the case where only eigenvalues are computed, but on more than one processor. Thus, in this case *m* refers to the number of eigenvalues computed on this processor.
3. The arrays *w* and *z* might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

Input Parameters

jobz

CHARACTER*1

= 'N': Compute eigenvalues only;

= 'V': Compute eigenvalues and eigenvectors.

range

CHARACTER*1

= 'A': all eigenvalues will be found.

= 'V': all eigenvalues in the half-open interval (*vl*,*vu*] will be found.

= 'I': the *il*-th through *iu*-th eigenvalues (of the entire matrix) will be found.

n

INTEGER

The order of the matrix. $n \geq 0$.

d

REAL for sstegr2

DOUBLE PRECISION for dstegr2

Array of size *n*

On entry, the *n* diagonal elements of the tridiagonal matrix *T*. Overwritten on exit.

e

REAL for sstegr2

DOUBLE PRECISION for dstegr2

Array of size *n*

On entry, the (*n*-1) subdiagonal elements of the tridiagonal matrix *T* in elements 1 to *n*-1 of *e*. *e*(*n*) need not be set on input, but is used internally as workspace. Overwritten on exit.

vl

REAL for sstegr2

DOUBLE PRECISION for dstegr2

vu

REAL for sstegr2

DOUBLE PRECISION for dstegr2

If *range*='V', the lower and upper bounds of the interval to be searched for eigenvalues. *vl* < *vu*.

Not referenced if *range* = 'A' or 'I'.

il, iu

INTEGER

If *range*='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned.

$1 \le il \le iu \le n$, if $n > 0$.

Not referenced if *range* = 'A' or 'V'.

ldz INTEGER

The leading dimension of the array *z*. $ldz \ge 1$, and if *jobz* = 'V', then $ldz \ge \max(1,n)$.

nzc INTEGER

The number of eigenvectors to be held in the array *z*, storing the matrix *Z*.

If *range* = 'A', then $nzc \ge \max(1,n)$.

If *range* = 'V', then *nzc*≥ the number of eigenvalues in (*vl*,*vu*].

If *range* = 'I', then $nzc \ge iu-il+1$.

If *nzc* = -1, then a workspace query is assumed; the routine calculates the number of columns of the matrix *Z* that are needed to hold the eigenvectors. This value is returned as the first entry of the *z* array, and no error message related to *nzc* is issued.

lwork INTEGER

The size of the array *work*. $lwork \ge \max(1,18*n)$

if *jobz* = 'V', and $lwork \ge \max(1,12*n)$ if *jobz* = 'N'. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued.

liwork INTEGER

The size of the array *iwork*. $liwork \ge \max(1,10*n)$ if the eigenvectors are desired, and $liwork \ge \max(1,8*n)$ if only the eigenvalues are to be computed.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued.

dol, *dou* INTEGER

From the eigenvalues *w*(1:*m*), only eigenvectors *Z*(:,*dol*) to *Z*(:,*dou*) are computed.

If *dol* > 1, then *Z*(:,*dol*-1-*zoffset*) is used and overwritten.

If *dou* < *m*, then *Z*(:,*dou*+1-*zoffset*) is used and overwritten.

zoffset INTEGER

Offset for storing the eigenpairs when *z* is distributed in 1D-cyclic fashion

OUTPUT Parameters

m INTEGER

Globally summed over all processors, *m* equals the total number of eigenvalues found. 0 ≤*m*≤*n*. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu-il*+1. The local output equals *m* = *dou* - *dol* + 1.

w

REAL array of size *n* for sstegr2

DOUBLE PRECISION array of size *n* for dstegr2

Array of size *n*

The first *m* elements contain the selected eigenvalues in ascending order. Note that immediately after exiting this routine, only the eigenvalues from position *dol*:*dou* are reliable on this processor because the eigenvalue computation is done in parallel. Other processors will hold reliable information on other parts of the *w* array. This information is communicated in the ScaLAPACK driver.

z

REAL for sstegr2

DOUBLE PRECISION for dstegr2

Array of size (*ldz*, max(1,*m*)).

If *jobz* = 'V', and if *info* = 0, then the first *m* columns of the matrix *Z* stored in *z* contain some of the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the *i*-th column of *Z* holding the eigenvector associated with *w*(*i*).

If *jobz* = 'N', then *z* is not referenced.

Note: the user must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and can be computed with a workspace query by setting *nzc* = -1, see below.

isuppz

INTEGER array of size 2*max(1,*m*)

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th computed eigenvector is nonzero only in elements *isuppz*(2**i*-1) through *isuppz*(2**i*). This is relevant in the case when the matrix is split. *isuppz* is only set if *n*>2.

work

On exit, if *info* = 0, *work*(1) returns the optimal (and minimal) *lwork*.

iwork

On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*.

info

INTEGER

On exit, *info*

= 0: successful exit

other:if *info* = -*i*, the *i*-th argument had an illegal value

if *info* = 10X, internal error in ?larre2,

if *info* = 20X, internal error in ?larrv.

Here, the digit X = ABS(*iinfo*) < 10, where *iinfo* is the nonzero error code returned by ?larre2 or ?larrv, respectively.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?stegr2a

Computes selected eigenvalues and initial representations needed for eigenvector computations.

Syntax

```
call sstegr2a( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, work, lwork,
iwork, liwork, dol, dou, needil, neediu, inderr, nsplit, pivmin, scale, wl, wu, info )
```

```
call dstegr2a( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, work, lwork,
iwork, liwork, dol, dou, needil, neediu, inderr, nsplit, pivmin, scale, wl, wu, info )
```

Description

?stegr2a computes selected eigenvalues and initial representations needed for eigenvector computations in ?stegr2b. It is invoked in the ScaLAPACK MRRR driver p?syevr and the corresponding Hermitian version when both eigenvalues and eigenvectors are computed in parallel on multiple processors. For this case, ?stegr2a implements the first part of the MRRR algorithm, parallel eigenvalue computation and finding the root RRR. At the end of ?stegr2a, other processors might have a part of the spectrum that is needed to continue the computation locally. Once this eigenvalue information has been received by the processor, the computation can then proceed by calling the second part of the parallel MRRR algorithm, ?stegr2b.

Please note:

- The calling sequence has two additional integer parameters, (compared to LAPACK's stegr), these are *dol* and *dou* and should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant for the case *jobz* = 'V'.

  Globally invoked over all processors, ?stegr2a computes all the eigenvalues specified by *range*.

  ?stegr2a locally only computes the eigenvalues corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenvectors belonging to *w*(1) through *w*(*m*), only the eigenvectors belonging to eigenvalues *w*(*dol*) through *w*(*dou*) are computed. In this case, only the eigenvalues *dol*:*dou* are guaranteed to be fully accurate.
- *m* is not the number of eigenvalues specified by *range*, but it is *m* = *dou* - *dol* + 1. Instead, *m* refers to the number of eigenvalues computed on this processor.
- While no eigenvectors are computed in ?stegr2a itself (this is done later in ?stegr2b), the interface

  If *jobz* = 'V' then, depending on *range* and *dol*, *dou*, ?stegr2a might need more workspace in *z* then the original ?stegr. In particular, the arrays *w* and *z* might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Input Parameters

jobz — CHARACTER*1

= 'N': Compute eigenvalues only;

= 'V': Compute eigenvalues and eigenvectors.

range — CHARACTER*1

= 'A': all eigenvalues will be found.

= 'V': all eigenvalues in the half-open interval (*vl*,*vu*] will be found.

= 'I': the *il*-th through *iu*-th eigenvalues (of the entire matrix) will be found.

n INTEGER

The order of the matrix. $n \geq 0$.

d REAL for sstegr2a

DOUBLE PRECISION for dstegr2a

Array of size *n*

The *n* diagonal elements of the tridiagonal matrix *T*. Overwritten on exit.

e REAL for sstegr2a

DOUBLE PRECISION for dstegr2a

Array of size *n*

On entry, the (*n*-1) subdiagonal elements of the tridiagonal matrix *T* in elements 1 to *n*-1 of *e*. *e*(*n*) need not be set on input, but is used internally as workspace. Overwritten on exit.

vl, *vu* REAL for sstegr2a

DOUBLE PRECISION for dstegr2a

If *range*='V', the lower and upper bounds of the interval to be searched for eigenvalues. *vl* < *vu*.

Not referenced if *range* = 'A' or 'I'.

il, *iu* INTEGER

If *range*='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq il \leq iu \leq n$, if $n > 0$.

Not referenced if *range* = 'A' or 'V'.

ldz INTEGER

The leading dimension of the array *z*. $ldz \geq 1$, and if *jobz* = 'V', then $ldz \geq \max(1,n)$.

nzc INTEGER

The number of eigenvectors to be held in the array *z*.

If *range* = 'A', then $nzc \geq \max(1,n)$.

If *range* = 'V', then *nzc*≥ the number of eigenvalues in (*vl*,*vu*].

If *range* = 'I', then $nzc \geq iu-il+1$.

If *nzc* = -1, then a workspace query is assumed; the routine calculates the number of columns of the array *z* that are needed to hold the eigenvectors. This value is returned as the first entry of the *z* array, and no error message related to *nzc* is issued.

lwork INTEGER

The size of the array *work*. *lwork*≥ max(1,18**n*) if *jobz* = 'V', and *lwork*≥ max(1,12**n*) if *jobz* = 'N'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued.

liwork INTEGER

The size of the array *iwork*. *liwork*≥ max(1,10**n*) if the eigenvectors are desired, and *liwork*≥ max(1,8**n*) if only the eigenvalues are to be computed.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued.

dol, *dou* INTEGER

From all the eigenvalues *w*(1:*m*), only eigenvalues *w*(*dol*:*dou*) are computed.

OUTPUT Parameters

m INTEGER

Globally summed over all processors, *m* equals the total number of eigenvalues found. 0 ≤*m*≤*n*.

If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

The local output equals *m* = *dou* - *dol* + 1.

w REAL for sstegr2a

DOUBLE PRECISION for dstegr2a

Array of size *n*

The first *m* elements contain approximations to the selected eigenvalues in ascending order. Note that immediately after exiting this routine, only the eigenvalues from position *dol*:*dou* are reliable on this processor because the eigenvalue computation is done in parallel. The other entries outside *dol*:*dou*are very crude preliminary approximations. Other processors hold reliable information on these other parts of the *w* array.

This information is communicated in the ScaLAPACK driver.

z REAL for sstegr2a

DOUBLE PRECISION for dstegr2a

Array of size (*ldz*, max(1,*m*)).

?stegr2a does not compute eigenvectors, this is done in ?stegr2b. The argument *z* as well as all related other arguments only appear to keep the interface consistent and to signal to the user that this subroutine is meant to be used when eigenvectors are computed.

work On exit, if *info* = 0, *work*(1) returns the optimal (and minimal) *lwork*.

iwork — On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*.

needil, *neediu* — INTEGER

The indices of the leftmost and rightmost eigenvalues needed to accurately compute the relevant part of the representation tree. This information can be used to find out which processors have the relevant eigenvalue information needed so that it can be communicated.

inderr — INTEGER

inderr points to the place in the work space where the eigenvalue uncertainties (errors) are stored.

nsplit — INTEGER

The number of blocks into which *T* splits. 1 ≤*nsplit*≤*n*.

pivmin — REAL for sstegr2a

DOUBLE PRECISION for dstegr2a

The minimum pivot in the sturm sequence for *T*.

scale — REAL for sstegr2a

DOUBLE PRECISION for dstegr2a

The scaling factor for the tridiagonal *T*.

wl, *wu* — REAL for sstegr2a

DOUBLE PRECISION for dstegr2a

The interval (*wl*, *wu*] contains all the wanted eigenvalues.

It is either given by the user or computed in ?larre2a.

info — INTEGER

On exit, *info* = 0: successful exit

other: if *info* = -*i*, the *i*-th argument had an illegal value

if *info* = 10*x*, internal error in ?larre2a,

Here, the digit *x* = abs(*iinfo*) < 10, where *iinfo* is the nonzero error code returned by ?larre2a.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?stegr2b

From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.

Syntax

```
call sstegr2b( jobz, n, d, e, m, w, z, ldz, nzc, isuppz, work, lwork, iwork, liwork, dol,
dou, needil, neediu, indwlc, pivmin, scale, wl, wu, vstart, finish, maxcls, ndepth,
parity, zoffset, info )
```

```
call dstegr2b( jobz, n, d, e, m, w, z, ldz, nzc, isuppz, work, lwork, iwork, liwork, dol,
dou, needil, neediu, indwlc, pivmin, scale, wl, wu, vstart, finish, maxcls, ndepth,
parity, zoffset, info )
```

Description

?stegr2b should only be called after a call to ?stegr2a. From eigenvalues and initial representations computed by ?stegr2a, ?stegr2b computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors. It is potentially invoked multiple times on a given processor because the locally relevant representation tree might depend on spectral information that is "owned" by other processors and might need to be communicated.

Please note:

- The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant for the case *jobz* = 'V'. ?stegr2b only computes the eigenvectors corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenvectors belonging to *w*(1) through *w*(*m*), only the eigenvectors belonging to eigenvalues *w*(*dol*) through *w*(*dou*) are computed. In this case, only the eigenvalues *dol*:*dou* are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.
- The additional arguments *vstart*, *finish*, *ndepth*, *parity*, *zoffset* are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable *finish*equals .TRUE., the computation has ended. All eigenpairs between *dol* and *dou* have been computed. *m* is set to *dou* - *dol* + 1.
- ?stegr2b needs more workspace in *z* than the sequential ?stegr. It is used to store the conformal embedding of the local representation tree.

Input Parameters

jobz

CHARACTER*1

= 'N': Compute eigenvalues only;

= 'V': Compute eigenvalues and eigenvectors.

n

INTEGER

The order of the matrix. $n \geq 0$.

d

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

Array of size *n*

The *n* diagonal elements of the tridiagonal matrix T. Overwritten on exit.

e

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

Array of size *n*

The (*n*-1) subdiagonal elements of the tridiagonal matrix *T* in elements 1 to *n*-1 of *e*. *e*(*n*) need not be set on input, but is used internally as workspace. Overwritten on exit.

m INTEGER

The total number of eigenvalues found in ?stegr2a. 0 ≤*m*≤*n*.

w REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

Array of size *n*

The first *m* elements contain approximations to the selected eigenvalues in ascending order. Note that only the eigenvalues from the locally relevant part of the representation tree, that is all the clusters that include eigenvalues from *dol*:*dou*, are reliable on this processor. (It does not need to know about any others anyway.)

ldz INTEGER

The leading dimension of the array *z*. *ldz*≥ 1, and if *jobz* = 'V', then *ldz*≥ max(1,*n*).

nzc INTEGER

The number of eigenvectors to be held in the array *z*, storing the matrix *Z*.

lwork INTEGER

The size of the array *work*. *lwork*≥ max(1,18**n*)

if *jobz* = 'V', and *lwork*≥ max(1,12**n*) if *jobz* = 'N'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued.

liwork INTEGER

The size of the array *iwork*. *liwork*≥ max(1,10**n*) if the eigenvectors are desired, and *liwork*≥ max(1,8**n*) if only the eigenvalues are to be computed.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued.

dol, *dou* INTEGER

From the eigenvalues *w*(1:*m*), only eigenvectors *Z*(:,*dol*) to *Z*(:,*dou*) are computed.

If *dol* > 1, then *Z*(:,*dol*-1-*zoffset*) is used and overwritten.

If *dou* < *m*, then *Z*(:,*dou*+1-*zoffset*) is used and overwritten.

needil, *neediu* INTEGER

Describes which are the left and right outermost eigenvalues still to be computed. Initially computed by ?larre2a, modified in the course of the algorithm.

pivmin

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

The minimum pivot in the sturm sequence for *T*.

scale

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

The scaling factor for *T*. Used for unscaling the eigenvalues at the very end of the algorithm.

wl, *wu*

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

The interval (*wl*, *wu*] contains all the wanted eigenvalues.

vstart

LOGICAL

.TRUE. on initialization, set to .FALSE. afterwards.

finish

LOGICAL

Indicates whether all eigenpairs have been computed.

maxcls

INTEGER

The largest cluster worked on by this processor in the representation tree.

ndepth

INTEGER

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

parity

INTEGER

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

zoffset

INTEGER

Offset for storing the eigenpairs when *z* is distributed in 1D-cyclic fashion.

OUTPUT Parameters

z

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

Array of size (*ldz*, max(1,*m*))

If *jobz* = 'V', and if *info* = 0, then a subset of the first *m* columns of the matrix *Z*, stored in *z*, contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *Z* holding the eigenvector associated with *w*(*i*).

See *dol*, *dou* for more information.

isuppz

INTEGER array of size 2*max(1,*m*).

	The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th computed eigenvector is nonzero only in elements *isuppz*(2**i*-1) through *isuppz*(2**i*). This is relevant in the case when the matrix is split. *isuppz* is only set if *n*>2.
work	On exit, if *info* = 0, *work*(1) returns the optimal (and minimal) *lwork*.
iwork	On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*.
needil, *neediu*	Modified in the course of the algorithm.

indwlc	`REAL` for `sstegr2b`
	`DOUBLE PRECISION` for `dstegr2b`
	Pointer into the workspace location where the local eigenvalue representations are stored. ("Local eigenvalues" are those relative to the individual shifts of the RRRs.)

vstart	`.TRUE.` on initialization, set to `.FALSE.` afterwards.
finish	Indicates whether all eigenpairs have been computed
maxcls	The largest cluster worked on by this processor in the representation tree.
ndepth	The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.
parity	An internal parameter needed for the storage of the clusters on the current level of the representation tree.
info	`INTEGER` On exit, *info* = 0: successful exit other:if *info* = -*i*, the *i*-th argument had an illegal value if *info* = 20*x*, internal error in `?larrv2`. Here, the digit *x* = abs(`iinfo`) < 10, where `iinfo` is the nonzero error code returned by `?larrv2`

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?stein2

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.

Syntax

```
call sstein2(n, d, e, m, w, iblock, isplit, orfac, z, ldz, work, iwork, ifail, info)
call dstein2(n, d, e, m, w, iblock, isplit, orfac, z, ldz, work, iwork, ifail, info)
```

Description

The `?stein2`routine is a modified LAPACK routine `?stein`. It computes the eigenvectors of a real symmetric tridiagonal matrix *T* corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter *maxits* (currently set to 5).

Input Parameters

n

INTEGER. The order of the matrix T ($n \geq 0$).

m

INTEGER. The number of eigenvectors to be found ($0 \leq m \leq n$).

d, *e*, *w*

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays:

d(*), of size *n*. The *n* diagonal elements of the tridiagonal matrix *T*.

e(*), of size *n*.

The (*n*-1) subdiagonal elements of the tridiagonal matrix *T*, in elements 1 to *n*-1. *e*(*n*) need not be set.

w(*), of size *n*.

The first *m* elements of *w* contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array *w* from ?stebz with ORDER = 'B' is expected here).

The size of *w* must be at least max(1, *n*).

iblock

INTEGER.

Array of size *n*.

The submatrix indices associated with the corresponding eigenvalues in *w*;

iblock(*i*) = 1, if eigenvalue *w*(*i*) belongs to the first submatrix from the top,

iblock(*i*) = 2, if eigenvalue *w*(*i*) belongs to the second submatrix, etc. (The output array *iblock* from ?stebz is expected here).

isplit

INTEGER.

Array of size *n*.

The splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second submatrix consists of rows/columns *isplit*(1)+1 through *isplit*(2), etc. (The output array *isplit* from ?stebz is expected here).

orfac

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

orfac specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues which are within *orfac**||*T*|| of each other are to be orthogonalized.

ldz

INTEGER. The leading dimension of the output array *z*; *ldz* ≥ max(1, *n*).

work

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Workspace array of size $5n$.

iwork INTEGER. Workspace array of size n.

Output Parameters

z REAL for sstein2

DOUBLE PRECISION for dstein2

Array of size *ldz* by m.

The computed eigenvectors. The eigenvector associated with the eigenvalue $w(i)$ is stored in the *i*-th column of z. Any vector that fails to converge is set to its current iterate after *maxits* iterations.

ifail INTEGER.

Array of size m.

On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after *maxits* iterations, then their indices are stored in the array *ifail*.

info INTEGER.

info = 0, the exit is successful.

info < 0: if *info* = -*i*, the *i*-th had an illegal value.

info > 0: if *info* = *i*, then *i* eigenvectors failed to converge in *maxits* iterations. Their indices are stored in the array *ifail*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?dbtf2

Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).

Syntax

```
call sdbtf2(m, n, kl, ku, ab, ldab, info)
call ddbtf2(m, n, kl, ku, ab, ldab, info)
call cdbtf2(m, n, kl, ku, ab, ldab, info)
call zdbtf2(m, n, kl, ku, ab, ldab, info)
```

Description

The ?dbtf2routine computes an *LU* factorization of a general real/complex *m*-by-*n* band matrix *A* without using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling BLAS Routines and Functions.

Input Parameters

m INTEGER. The number of rows of the matrix *A* ($m \geq 0$).

n INTEGER. The number of columns in *A* (*n* ≥ 0).

kl INTEGER. The number of sub-diagonals within the band of *A* (*kl* ≥ 0).

ku INTEGER. The number of super-diagonals within the band of *A* (*ku* ≥ 0).

ab REAL for sdbtf2

DOUBLE PRECISION for ddbtf2

COMPLEX for cdbtf2

COMPLEX*16 for zdbtf2.

Array of size *ldab* by *n*.

The matrix *A* in band storage, in rows *kl*+1 to 2*kl*+*ku*+1; rows 1 to *kl* of the array need not be set. The *j*-th column of *A* is stored in the*j*-th column of the array *ab* as follows: *ab*(*kl*+*ku*+1+*i*-*j*,*j*) = *A*(*i*,*j*) for *max*(1,*j*-*ku*) ≤ *i* ≤ *min*(*m*,*j*+*kl*).

ldab INTEGER. The leading dimension of the array *ab*.

(*ldab* ≥ 2*kl* + *ku* +1)

Output Parameters

ab On exit, details of the factorization: *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1. See the *Application Notes* below for further details.

info INTEGER.

= 0: successful exit

< 0: if *info* = - *i*, the *i*-th argument had an illegal value,

> 0: if *info* = + *i*, *U*(*i*,*i*) is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when *m* = *n* = 6, *kl* = 2, *ku* = 1:

on entry

$$\begin{bmatrix} * & a12 & a23 & a34 & a45 & a56 \\ a11 & a22 & a33 & a44 & a55 & a66 \\ a21 & a32 & a43 & a54 & a65 & * \\ a31 & a42 & a53 & a64 & * & * \end{bmatrix}$$

on exit

$$\begin{bmatrix} * & u12 & u23 & u34 & u45 \\ u11 & u22 & u33 & u44 & u55 \\ m21 & m32 & m43 & m54 & m65 \\ m31 & m42 & m53 & m64 & * \end{bmatrix}$$

The routine does not use array elements marked *; elements marked + need not be set on entry, but the routine requires them to store elements of *U*, because of fill-in resulting from the row interchanges.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?dbtrf

Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdbtrf(m, n, kl, ku, ab, ldab, info)
call ddbtrf(m, n, kl, ku, ab, ldab, info)
call cdbtrf(m, n, kl, ku, ab, ldab, info)
call zdbtrf(m, n, kl, ku, ab, ldab, info)
```

Description

This routine computes an LU factorization of a real *m*-by-*n* band matrix *A* without using partial pivoting or row interchanges.

This is the blocked version of the algorithm, calling BLAS Routines and Functions.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq 0$).
kl	INTEGER. The number of sub-diagonals within the band of *A* ($kl \geq 0$).
ku	INTEGER. The number of super-diagonals within the band of *A* ($ku \geq 0$).
ab	REAL for sdbtrf DOUBLE PRECISION for ddbtrf COMPLEX for cdbtrf COMPLEX*16 for zdbtrf. Array of size *ldab* by *n*. The matrix *A* in band storage, in rows *kl*+1 to 2*kl*+*ku*+1; rows 1 to *kl* of the array need not be set. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows: *ab*(*kl*+*ku*+1+*i*–*j*,*j*) = *A*(*i*,*j*) for max(1,*j*–*ku*) ≤ *i* ≤ min(*m*,*j*+*kl*).
ldab	INTEGER. The leading dimension of the array *ab*. (*ldab* ≥ 2*kl* + *ku* +1)

Output Parameters

ab	On exit, details of the factorization: *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1. See the *Application Notes* below for further details.
info	INTEGER. = 0: successful exit < 0: if *info* = - *i*, the *i*-th argument had an illegal value,

> 0: if *info* = + *i*, *U*(*i*,*i*) is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when *m* = *n* = 6, *kl* = 2, *ku* = 1:

on entry

$$\begin{bmatrix} * & a12 & a23 & a34 & a45 & a56 \\ a11 & a22 & a33 & a44 & a55 & a66 \\ a21 & a32 & a43 & a54 & a65 & * \\ a31 & a42 & a53 & a64 & * & * \end{bmatrix}$$

on exit

$$\begin{bmatrix} * & u12 & u23 & u34 & u45 \\ u11 & u22 & u33 & u44 & u55 \\ m21 & m32 & m43 & m54 & m65 \\ m31 & m42 & m53 & m64 & * \end{bmatrix}$$

The routine does not use array elements marked *.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?dttrf

Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdttrf(n, dl, d, du, info)
call ddttrf(n, dl, d, du, info)
call cdttrf(n, dl, d, du, info)
call zdttrf(n, dl, d, du, info)
```

Description

The ?dttrfroutine computes an *LU* factorization of a real or complex tridiagonal matrix *A* using elimination without partial pivoting.

The factorization has the form *A* = *L***U*, where *L* is a product of unit lower bidiagonal matrices and *U* is upper triangular with nonzeros only in the main diagonal and first superdiagonal.

Input Parameters

n

INTEGER. The order of the matrix *A* (*n* ≥ 0).

dl, *d*, *du*

REAL for sdttrf

DOUBLE PRECISION for ddttrf

COMPLEX for cdttrf

COMPLEX*16 for zdttrf.

Arrays containing elements of *A*.

The array *dl* of size (*n*–1) contains the sub-diagonal elements of *A*.

The array *d* of size *n* contains the diagonal elements of *A*.

The array *du* of size (*n*-1) contains the super-diagonal elements of *A*.

Output Parameters

dl — Overwritten by the (*n*-1) multipliers that define the matrix *L* from the *LU* factorization of *A*.

d — Overwritten by the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

du — Overwritten by the (*n*-1) elements of the first super-diagonal of *U*.

info — INTEGER.

= 0: successful exit

< 0: if *info* = - *i*, the *i*-th argument had an illegal value,

> 0: if *info* = *i*, *U*(*i*,*i*) is exactly 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?dttrsv

Solves a general tridiagonal system of linear equations using the LU factorization computed by ?dttrf.

Syntax

```
call sdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call ddttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call cdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call zdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
```

Description

The ?dttrsvroutine solves one of the following systems of linear equations:

$L*X = B$, $L^T*X = B$, or $L^H*X = B$,

$U*X = B$, $U^T*X = B$, or $U^H*X = B$

with factors of the tridiagonal matrix *A* from the *LU* factorization computed by ?dttrf.

Input Parameters

uplo — CHARACTER*1.

Specifies whether to solve with *L* or *U*.

trans — CHARACTER. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', then $A*X=B$ is solved for *X* (no transpose).

If *trans* = 'T', then $A^T*X = B$ is solved for *X* (transpose).

	If *trans* = 'C', then $A^H*X = B$ is solved for *X* (conjugate transpose).
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides, that is, the number of columns in the matrix *B* (*nrhs* ≥ 0).
dl,*d*,*du*,*b*	REAL for sdttrsv DOUBLE PRECISION for ddttrsv COMPLEX for cdttrsv COMPLEX*16 for zdttrsv. The array *dl* of size (*n* - 1) contains the (*n* - 1) multipliers that define the matrix *L* from the *LU* factorization of *A*. The array *d* of size *n* contains *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*. The array *du* of size (*n* - 1) contains the (*n* - 1) elements of the first super-diagonal of *U*. On entry, the array *b* of size (*ldb*,*nrhs*) contains the right-hand side of matrix *B*.
ldb	INTEGER. The leading dimension of the array *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?pttrsv

*Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the L*D*L^H factorization computed by* ?pttrf.

Syntax

```
call spttrsv(trans, n, nrhs, d, e, b, ldb, info)
call dpttrsv(trans, n, nrhs, d, e, b, ldb, info)
call cpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
call zpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
```

Description

The ?pttrsvroutine solves one of the triangular systems:

$L^T*X = B$, or $L*X = B$ for real flavors,

or

$L*X = B$, or $L^H*X = B$,

*U*X = B*, or $U^H*X = B$ for complex flavors,

where *L* (or *U* for complex flavors) is the Cholesky factor of a Hermitian positive-definite tridiagonal matrix *A* such that

$A = L*D*L^H$ (computed by spttrf/dpttrf)

or

$A = U^H*D*U$ or $A = L*D*L^H$ (computed by cpttrf/zpttrf).

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix *A* is stored and the form of the factorization:

If *uplo* = 'U', *e* is the superdiagonal of *U*, and $A = U^H*D*U$ or $A = L*D*L^H$;

if *uplo* = 'L', *e* is the subdiagonal of *L*, and $A = L*D*L^H$.

The two forms are equivalent, if *A* is real.

trans

CHARACTER.

Specifies the form of the system of equations:

for real flavors:

if *trans* = 'N': *L*X = B* (no transpose)

if *trans* = 'T': $L^T*X = B$ (transpose)

for complex flavors:

if *trans* = 'N': *U*X = B* or *L*X = B* (no transpose)

if *trans* = 'C': $U^H*X = B$ or $L^H*X = B$ (conjugate transpose).

n

INTEGER. The order of the tridiagonal matrix *A*. $n \geq 0$.

nrhs

INTEGER. The number of right hand sides, that is, the number of columns of the matrix *B*. $nrhs \geq 0$.

d

REAL array of size *n*. The *n* diagonal elements of the diagonal matrix *D* from the factorization computed by ?pttrf.

e

COMPLEX array of size (*n*–1). The (*n*-1) off-diagonal elements of the unit bidiagonal factor *U* or *L* from the factorization computed by ?pttrf. See *uplo*.

b

COMPLEX array of size *ldb*by *nrhs*.

On entry, the right hand side matrix *B*.

ldb

INTEGER.

The leading dimension of the array *b*.

$ldb \geq \max(1, n)$.

Output Parameters

b	On exit, the solution matrix *X*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?steqr2

Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

Syntax

```
call ssteqr2(compz, n, d, e, z, ldz, nr, work, info)
call dsteqr2(compz, n, d, e, z, ldz, nr, work, info)
```

Description

The ?steqr2routine is a modified version of LAPACK routine ?steqr. The ?steqr2routine computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. ?steqr2 is modified from ?steqr to allow each ScaLAPACK process running ?steqr2 to perform updates on a distributed matrix Q. Proper usage of ?steqr2 can be gleaned from examination of ScaLAPACK routine p?syev.

Input Parameters

compz	CHARACTER*1. Must be 'N' or 'I'. If *compz* = 'N', the routine computes eigenvalues only. If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix *T*. *z* must be initialized to the identity matrix by p?laset or ?laset prior to entering this subroutine.
n	INTEGER. The order of the matrix *T* (*n* ≥ 0).
d, *e*, *work*	REAL for ssteqr2 DOUBLE PRECISION for dsteqr2 Arrays: *d* contains the diagonal elements of *T*. The size of *d* must be at least max(1, *n*). *e* contains the (*n*-1) subdiagonal elements of *T*. The size of *e* must be at least max(1, *n*-1). *work* is a workspace array. The size of *work* is max(1, 2*n-2). If *compz* = 'N', then *work* is not referenced.
z	(local)

REAL for ssteqr2

DOUBLE PRECISION for dsteqr2

Array of global size *n*by *n* and of local size *ldz*by *nr*.

If *compz* = 'V', then *z* contains the orthogonal matrix used in the reduction to tridiagonal form.

ldz INTEGER. The leading dimension of the array *z*. Constraints:

ldz ≥ 1,

ldz ≥ max(1, *n*), if eigenvectors are desired.

nr INTEGER. *nr* = max(1, *numroc*(*n*, *nb*, *myprow*, 0, *nprocs*)).

If *compz* = 'N', then *nr* is not referenced.

Output Parameters

d On exit, the eigenvalues in ascending order, if *info* = 0.

See also *info*.

e On exit, *e* has been destroyed.

z On exit, if *info* = 0, then,

if *compz* = 'V', *z* contains the orthonormal eigenvectors of the original symmetric matrix, and if *compz* = 'I', *z* contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If *compz* = 'N', then *z* is not referenced.

info INTEGER.

info = 0, the exit is successful.

info < 0: if *info* = -*i*, the *i*-th had an illegal value.

info > 0: the algorithm has failed to find all the eigenvalues in a total of 30*n* iterations;

if *info* = *i*, then *i* elements of *e* have not converged to zero; on exit, *d* and *e* contain the elements of a symmetric tridiagonal matrix, which is orthogonally similar to the original matrix.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?trmvt

Performs matrix-vector operations.

Syntax

```
call strmvt (uplo, n, t, ldt, x, incx, y, incy, w, incw, z, incz )
call dtrmvt (uplo, n, t, ldt, x, incx, y, incy, w, incw, z, incz )
call ctrmvt (uplo, n, t, ldt, x, incx, y, incy, w, incw, z, incz )
call ztrmvt (uplo, n, t, ldt, x, incx, y, incy, w, incw, z, incz )
```

Description

?trmvt performs the matrix-vector operations as follows:

strmvt and dtrmvt: x := *T*' *y, and w := *T* *z

ctrmvt and ztrmvt: x := conjg(*T*') *y, and w := *T* *z,

where x is an *n* element vector and *T* is an *n*-by-*n* upper or lower triangular matrix.

Input Parameters

uplo

CHARACTER*1.

On entry, *uplo* specifies whether the matrix is an upper or lower triangular matrix as follows:

uplo = 'U' or 'u'

A is an upper triangular matrix.

uplo = 'L' or 'l'

A is a lower triangular matrix.

Unchanged on exit.

n

INTEGER.

On entry, *n* specifies the order of the matrix *A*. *n* must be at least zero.

Unchanged on exit.

t

REAL for strmvt

DOUBLE PRECISION for dtrmvt

COMPLEX for ctrmvt

DOUBLE COMPLEX for ztrmvt

Array of size (*ldt*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *t* must contain the upper triangular matrix and the strictly lower triangular part of *t* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *t* must contain the lower triangular matrix and the strictly upper triangular part of *t* is not referenced.

ldt

INTEGER.

On entry, *lda* specifies the first dimension of *A* as declared in the calling (sub) program. *lda* must be at least max(1, *n*).

Unchanged on exit.

incx

INTEGER.

On entry, *incx* specifies the increment for the elements of *x*. *incx* must not be zero.

Unchanged on exit.

y

REAL for strmvt

DOUBLE PRECISION for dtrmvt

COMPLEX for ctrmvt

DOUBLE COMPLEX for ztrmvt

Array of size at least (1 + (*n* - 1)*abs(*incy*)).

Before entry, the incremented array *y* must contain the *n* element vector *y*.

Unchanged on exit.

incy INTEGER.

On entry, *incy* specifies the increment for the elements of *y*. *incy* must not be zero.

Unchanged on exit.

incw INTEGER.

On entry, *incw* specifies the increment for the elements of *w*. *incw* must not be zero.

Unchanged on exit.

z REAL for strmvt

DOUBLE PRECISION for dtrmvt

COMPLEX for ctrmvt

DOUBLE COMPLEX for ztrmvt

Array of size at least (1 + (*n* - 1)*abs(*incz*)).

Before entry, the incremented array *z* must contain the *n* element vector *z*.

Unchanged on exit.

incz INTEGER.

On entry, *incz* specifies the increment for the elements of *z*. *incz* must not be zero.

Unchanged on exit.

Output Parameters

t Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *t* must contain the upper triangular matrix and the strictly lower triangular part of *t* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *t* must contain the lower triangular matrix and the strictly upper triangular part of *t* is not referenced.

x REAL for strmvt

DOUBLE PRECISION for dtrmvt

COMPLEX for ctrmvt

DOUBLE COMPLEX for ztrmvt

Array of size at least (1 + (*n* - 1)*abs(*incx*)).

On exit, *x* = *T*' * *y*.

w

REAL for strmvt

DOUBLE PRECISION for dtrmvt

COMPLEX for ctrmvt

DOUBLE COMPLEX for ztrmvt

Array of size at least (1 + (*n* - 1)*abs(*incw*)).

On exit, *w* = *T* * *z*.

pilaenv

Returns the positive integer value of the logical blocking size.

Syntax

```
value = pilaenv (ictxt, prec )
```

Include Files

- mkl_pblas.h

Description

pilaenv returns the positive integer value of the logical blocking size. This value is machine and precision specific. This version provides a logical blocking size which should give good though not optimal performance on many of the currently available distributed-memory concurrent computers. You are encouraged to modify this subroutine to set this tuning parameter for your particular machine.

Input Parameters

ictxt

INTEGER. On entry, *ictxt* specifies the BLACS context handle, indicating the global context of the operation. The context itself is global, but the value of *ictxt* is local.

prec

CHARACTER*1. On input, *prec* specifies the precision for which the logical block size should be returned as follows:

prec = 'S' or 's' single precision real,

prec = 'D' or 'd' double precision real,

prec = 'C' or 'c' single precision complex,

prec = 'Z' or 'z' double precision complex,

prec = 'I' or 'i' integer.

Application Notes

Before modifying this routine to tune the library performance on your system, be aware of the following:

1. The value this function returns must be strictly larger than zero,
2. If you are planning to link your program with different instances of the library (for example, on a heterogeneous machine), you *must* compile each instance of the library with exactly the same version of this routine for obvious interoperability reasons.

pilaenvx

Called from the ScaLAPACK routines to choose problem-dependent parameters for the local environment.

Syntax

```
result = pilaenvx (ictxt, ispec, name, opts, n1, n2, n3, n4 )
```

Include Files

- mkl.fi

Description

pilaenvx is called from the ScaLAPACK routines to choose problem-dependent parameters for the local environment. See *ispec* for a description of the parameters. This version provides a set of parameters which should give good, though not optimal, performance on many of the currently available computers. You are encouraged to modify this subroutine to set the tuning parameters for your particular machine using the option and problem size information in the arguments.

Input Parameters

ictxt

(local input) INTEGER. On entry, *ictxt* specifies the BLACS context handle, indicating the global context of the operation. The context itself is global, but the value of *ictxt* is local.

ispec

(global input) INTEGER.

Specifies the parameter to be returned as the value of pilaenvx.

= 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance (unlikely).

= 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used.

= 3: the crossover point (in a block routine, for N less than this value, an unblocked routine should be used).

= 4: the number of shifts, used in the nonsymmetric eigenvalue routines (DEPRECATED).

= 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least *k* by *m*, where *k* is given by pilaenvx(2,...) and *m* by pilaenvx(5,...).

= 6: the crossover point for the SVD (when reducing an *m* by *n* matrix to bidiagonal form, if max(*m*,*n*)/min(*m*,*n*) exceeds this value, a QR factorization is used first to reduce the matrix to a triangular form).

= 7: the number of processors.

= 8: the crossover point for the multishift QR method for nonsymmetric eigenvalue problems (DEPRECATED).

= 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by ?gelsd and ?gesdd).

=10: IEEE NaN arithmetic can be trusted not to trap.

=11: infinity arithmetic can be trusted not to trap.

	12 <= *ispec* <= 16: `p?hseqr` or one of its subroutines, see `piparmq` for detailed explanation. 17 <= *ispec* <= 22: Parameters for `pb?trord`/`p?hseqr` (not all), as follows: =17: maximum number of concurrent computational windows; =18: number of eigenvalues/bulges in each window; =19: computational window size; =20: minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; =21: width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; =22: the maximum number of eigenvalues moved together over a process border; =23: the number of processors involved in Aggressive Early Deflation (AED); =99: Maximum iteration chunksize in OpenMP parallelization.
name	(global input) CHARACTER*(*). The name of the calling subroutine, in either upper case or lower case.
opts	(global input) CHARACTER*(*). The character options to the subroutine name, concatenated into a single character string. For example, *uplo* = 'U', *trans* = 'T', and *diag* = 'N' for a triangular routine would be specified as *opts* = 'UTN'.
n1, *n2*, *n3*, and *n4*	(global input) INTEGER. Problem dimensions for the subroutine name; these may not all be required.

Output Parameters

result	(global output) INTEGER. >= 0: the value of the parameter specified by *ispec*. < 0: if `pilaenvx` = -*k*, the *k*-th argument had an illegal value.

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine name, in the same order that they appear in the argument list for name, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1*, *n2*, *n3*, and *n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by `ilaenv` is checked for validity in the calling subroutine. For example, `ilaenv` is used to retrieve the optimal block size for `strtri` as follows:

```
NB = ilaenv( 1, 'STRTRI', UPLO // DIAG, N, -1, -1, -1 )
   IF( NB.LE.1 ) NB = MAX( 1, N )
```

The same conventions hold for this ScaLAPACK-style variant.

pjlaenv

Called from the ScaLAPACK symmetric and Hermitian tailored eigen-routines to choose problem-dependent parameters for the local environment.

Syntax

```
result = pjlaenv (ictxt, ispec, name, opts, n1, n2, n3, n4 )
```

Include Files

- mkl.fi

Description

`pjlaenv` is called from the ScaLAPACK symmetric and Hermitian tailored eigen-routines to choose problem-dependent parameters for the local environment. See *ispec* for a description of the parameters. This version provides a set of parameters which should give good, though not optimal, performance on many of the currently available computers. You are encouraged to modify this subroutine to set the tuning parameters for your particular machine using the option and problem size information in the arguments.

Input Parameters

ispec
(global input) INTEGER. Specifies the parameter to be returned as the value of `pjlaenv`.

= 1: the data layout blocksize;

= 2: the panel blocking factor;

= 3: the algorithmic blocking factor;

= 4: execution path control;

= 5: maximum size for direct call to the LAPACK routine.

name
(global input) CHARACTER*(*). The name of the calling subroutine, in either upper case or lower case.

opts
(global input) CHARACTER*(*). The character options to the subroutine name, concatenated into a single character string. For example, *uplo* = `'U'`, *trans* = `'T'`, and *diag* = `'N'` for a triangular routine would be specified as *opts* = `'UTN'`.

n1, *n2*, *n3*, and *n4*
(global input) INTEGER. Problem dimensions for the subroutine name; these may not all be required. At present, only *n1* is used, and it (*n1*) is used only for `'TTRD'`.

Output Parameters

result
(global or local output) INTEGER.

>= 0: the value of the parameter specified by *ispec*.

< 0: if `pjlaenv` = -*k*, the *k*-th argument had an illegal value. Most parameters set via a call to `pjlaenv` must be identical on all processors and hence `pjlaenv` will return the same value to all

procesors (i.e. global output). However some, in particular, the panel blocking factor can be different on each processor and hence pjlaenv can return different values on different processors (i.e. local output).

Application Notes

The following conventions have been used when calling pjlaenv from the ScaLAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine name, in the same order that they appear in the argument list for name, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1*, *n2*, *n3*, and *n4* are specified in the order that they appear in the argument list for name. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
   a. The parameter value returned by pjlaenv is checked for validity in the calling subroutine. For example, pjlaenv is used to retrieve the optimal blocksize for STRTRI as follows:

```
NB = pjlaenv( 1, 'STRTRI', UPLO // DIAG, N, -1, -1, -1 )
IF( NB.LE.1 ) NB = MAX( 1, N )
```

pjlaenv is patterned after ilaenv and keeps the same interface in anticipation of future needs, even though pjlaenv is only sparsely used at present in ScaLAPACK. Most ScaLAPACK codes use the input data layout blocking factor as the algorithmic blocking factor - hence there is no need or opportunity to set the algorithmic or data decomposition blocking factor. pXYYtevx.f and pXYYtgvx.f and pXYYttrd.f are the only codes which call pjlaenv. pXYYtevx.f and pXYYtgvx.f redistribute the data to the best data layout for each transformation. pXYYttrd.f uses a data layout blocking factor of 1.

Additional ScaLAPACK Routines

call pchettrd (*uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info*)

call pzhettrd (*uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info*)

call pslaed0 (*n, d, e, q, iq, jq, descq, work, iwork, info*)

call pdlaed0 (*n, d, e, q, iq, jq, descq, work, iwork, info*)

call pslaed1 (*n, n1, d, id, q, iq, jq, descq, rho, work, iwork, info*)

call pdlaed1 (*n, n1, d, id, q, iq, jq, descq, rho, work, iwork, info*)

call pslaed2 (*ictxt, k, n, n1, nb, d, drow, dcol, q, ldq, rho, z, w, dlamda, q2, ldq2, qbuf, ctot, psm, npcol, indx, indxc, indxp, indcol, coltyp, nn, nn1, nn2, ib1, ib2*)

call pdlaed2 (*ictxt, k, n, n1, nb, d, drow, dcol, q, ldq, rho, z, w, dlamda, q2, ldq2, qbuf, ctot, psm, npcol, indx, indxc, indxp, indcol, coltyp, nn, nn1, nn2, ib1, ib2*)

call pslaed3 (*ictxt, k, n, nb, d, drow, dcol, rho, dlamda, w, z, u, ldu, buf, indx, indcol, indrow, indxr, indxc, ctot, npcol, info*)

call pdlaed3 (*ictxt, k, n, nb, d, drow, dcol, rho, dlamda, w, z, u, ldu, buf, indx, indcol, indrow, indxr, indxc, ctot, npcol, info*)

call pslaedz (*n, n1, id, q, iq, jq, ldq, descq, z, work*)

call pdlaedz (*n, n1, id, q, iq, jq, ldq, descq, z, work*)

call pdlaiectb (*sigma, n, d, count*)

call pdlaiectl (*sigma, n, d, count*)

call slamov (*uplo, m, n, a, lda, b, ldb*)

call dlamov (*uplo, m, n, a, lda, b, ldb*)

```
call clamov (uplo, m, n, a, lda, b, ldb )
call zlamov (uplo, m, n, a, lda, b, ldb )
call pslamr1d (n, a, ia, ja, desca, b, ib, jb, descb )
call pdlamr1d (n, a, ia, ja, desca, b, ib, jb, descb )
call pclamr1d (n, a, ia, ja, desca, b, ib, jb, descb )
call pzlamr1d (n, a, ia, ja, desca, b, ib, jb, descb )
call clanv2 (a, b, c, d, rt1, rt2, cs, sn )
call zlanv2 (a, b, c, d, rt1, rt2, cs, sn )
call pclattrs (uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, info )
call pzlattrs (uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, info )
call pssyttrd (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info )
call pdsyttrd (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info )
integer function piparmq (ictxt, ispec, name, opts, n, ilo, ihi, lworknb )
```

For descriptions of these functions, please see http://www.netlib.org/scalapack/explore-html/files.html.

ScaLAPACK Utility Functions and Routines

This section describes ScaLAPACK utility functions and routines. Summary information about these routines is given in the following table:

ScaLAPACK Utility Functions and Routines

Routine Name	Data Types	Description
p?labad	s,d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
p?lachkieee	s,d	Performs a simple check for the features of the IEEE standard. (C interface function).
p?lamch	s,d	Determines machine parameters for floating-point arithmetic.
p?lasnbt	s,d	Computes the position of the sign bit of a floating-point number. (C interface function).
descinit	N/A	Initializes the array descriptor for distributed matrix.
numroc	N/A	Computes the number of rows or columns of a distributed matrix owned by the process.

See Also

pxerbla Error handling routine called by ScaLAPACK routines.

p?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call pslabad(ictxt, small, large)
call pdlabad(ictxt, small, large)
```

Description

The p?labadroutine takes as input the values computed by p?lamch for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by p?lamch. This subroutine is needed because p?lamch does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

In addition, this routine performs a global minimization and maximization on these values, to support heterogeneous computing networks.

Input Parameters

ictxt (global) INTEGER.

The BLACS context handle in which the computation takes place.

small (local).

REAL PRECISION for pslabad.

DOUBLE PRECISION for pdlabad.

On entry, the underflow threshold as computed by p?lamch.

large (local).

REAL PRECISION for pslabad.

DOUBLE PRECISION for pdlabad.

On entry, the overflow threshold as computed by p?lamch.

Output Parameters

small (local).

On exit, if *log10*(*large*) is sufficiently large, the square root of *small*, otherwise unchanged.

large (local).

On exit, if *log10*(*large*) is sufficiently large, the square root of *large*, otherwise unchanged.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lachkieee

Performs a simple check for the features of the IEEE standard. (C interface function).

Syntax

```
void pslachkieee(int *isieee, float *rmax, float *rmin);
void pdlachkieee(int *isieee, float *rmax, float *rmin);
```

Description

The p?lachkieeeroutine performs a simple check to make sure that the features of the IEEE standard are implemented. In some implementations, p?lachkieee may not return.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

rmax (local).

`REAL` for `pslachkieee`

`DOUBLE PRECISION` for `pdlachkieee`

The overflow threshold `(= ?lamch ('O'))`.

rmin (local).

`REAL` for `pslachkieee`

`DOUBLE PRECISION` for `pdlachkieee`

The underflow threshold `(= ?lamch ('U'))`.

Output Parameters

isieee (local). `INTEGER`.

On exit, *isieee* `= 1` implies that all the features of the IEEE standard that we rely on are implemented. On exit, *isieee* `= 0` implies that some the features of the IEEE standard that we rely on are missing.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

val `= pslamch(`*ictxt*`, `*cmach*`)`

val `= pdlamch(`*ictxt*`, `*cmach*`)`

Description

The `p?lamch`routine determines single precision machine parameters.

Input Parameters

ictxt (global). `INTEGER`. The BLACS context handle in which the computation takes place.

cmach (global) `CHARACTER*1`.

Specifies the value to be returned by `p?lamch`:

`= 'E'` or `'e'`, `p?lamch := eps`

`= 'S'` or `'s'` , `p?lamch := sfmin`

`= 'B'` or `'b'`, `p?lamch := base`

`= 'P'` or `'p'`, `p?lamch := eps*base`

= 'N' or 'n', p?lamch := t

= 'R' or 'r', p?lamch := rnd

= 'M' or 'm', p?lamch := emin

= 'U' or 'u', p?lamch := rmin

= 'L' or 'l', p?lamch := emax

= 'O' or 'o', p?lamch := rmax,

where

eps = relative machine precision

sfmin = safe minimum, such that 1/sfmin does not overflow

base = base of the machine

prec = eps*base

t = number of (base) digits in the mantissa

rnd = 1.0 when rounding occurs in addition, 0.0 otherwise

emin = minimum exponent before (gradual) underflow

rmin = underflow threshold - $base^{(emin-1)}$

emax = largest exponent before overflow

rmax = overflow threshold - $(base^{emax})*(1-eps)$

Output Parameters

val	Value returned by the routine.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lasnbt

Computes the position of the sign bit of a floating-point number. (C interface function).

Syntax

```
void pslasnbt(int *ieflag);
void pdlasnbt(int *ieflag);
```

Description

The p?lasnbtroutine finds the position of the signbit of a single/double precision floating point number. This routine assumes IEEE arithmetic, and hence, tests only the 32-nd bit (for single precision) or 32-nd and 64-th bits (for double precision) as a possibility for the signbit. sizeof(int) is assumed equal to 4 bytes.

If a compile time flag (NO_IEEE) indicates that the machine does not have IEEE arithmetic, *ieflag* = 0 is returned.

Output Parameters

ieflag	INTEGER.

This flag indicates the position of the signbit of any single/double precision floating point number.

ieflag = 0, if the compile time flag NO_IEEE indicates that the machine does not have IEEE arithmetic, or if sizeof(int) is different from 4 bytes.

ieflag = 1 indicates that the signbit is the 32-nd bit for a single precision routine.

In the case of a double precision routine:

ieflag = 1 indicates that the signbit is the 32-nd bit (Big Endian).

ieflag = 2 indicates that the signbit is the 64-th bit (Little Endian).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

descinit

Initializes the array descriptor for distributed matrix.

Syntax

```
call descinit( desc, m, n, mb, nb, irsrc, icsrc, ictxt, lld, info )
```

Description

The descintroutine initializes the array descriptor for distributed matrix.

Input Parameters

desc (global) INTEGER. array of dimension DLEN_. The array descriptor of a distributed matrix to be set.

m (global input)INTEGER. The number of rows in the distributed matrix. M >=0.

n (global input) INTEGER. The number of columns in the distributed matrix. N >=0.

mb (global input) INTEGER. The blocking factor used to distribute the rows of the matrix. MB >= 1.

nb (global input) INTEGER.INTEGER. The blocking factor used to distribute the columns of the matrix. NB >= 1.

lrsrc (global input) INTEGER. The process row over which the first row of the matrix is distributed. 0 <= IRSRC < NPROW.

lcsrc (global input) INTEGER. The process column over which the first column of the matrix is distributed. 0 <= ICSRC < NPCOL.

ictxt (global input) INTEGER. The BLACS context handle, indicating the global context of the operation on the matrix. The context itself is global.

lld (local input) INTEGER. The leading dimension of the local array storing the local blocks of the distributed matrix. LLD >= MAX(1,LOCr(M)). LOCr() denotes the number of rows of a global dense matrix that the process in a grid receives after data distributing.

Output Parameters

info	(output)INTEGER. = 0: successful exit < 0: if INFO = -i, the i-th argument had an illegal value

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

numroc

Computes the number of rows or columns of a distributed matrix owned by the process.

Syntax

```
val = numroc( n, nb, iproc, srcproc, nprocs )
```

Description

The numrocroutine computes the number of rows or columns of a distributed matrix owned by the process.

Input Parameters

n	(global input)INTEGER. The number of rows/columns in distributed matrix.
nb	(global input)INTEGER. Block size, size of the blocks the distributed matrix is split into.
iproc	(local input)INTEGER. The coordinate of the process whose local array row or column is to be determined.
srcproc	(global input)INTEGER. The coordinate of the process that possesses the first row or column of the distributed matrix.
nprocs	(global input)INTEGER. The total number processes over which the matrix is distributed.

Output Parameters

info	(output)INTEGER. Value returned by the function.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

ScaLAPACK Redistribution/Copy Routines

This section describes ScaLAPACK redistribution/copy routines. Summary information about these routines is given in the following table:

ScaLAPACK Redistribution/Copy Routines

Routine Name	Data Types	Description
p?gemr2d	s,d,c,z,i	Copies a submatrix from one general rectangular matrix to another.
p?trmr2d	s,d,c,z,i	Copies a submatrix from one trapezoidal matrix to another.

See Also

pxerbla Error handling routine called by ScaLAPACK routines.

p?gemr2d

Copies a submatrix from one general rectangular matrix to another.

Syntax

```
call psgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pdgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pcgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pzgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pigemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
```

Description

The p?gemr2droutine copies the indicated matrix or submatrix of *A* to the indicated matrix or submatrix of *B*. It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With p?trmr2d, these routines are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix *A* in context *A* (distributed over process grid *A*) and copy it to a matrix or submatrix *B* in context *B* (distributed over process grid *B*).

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context *A* is disjoint from context *B*. The general rules for which parameters need to be set are:

- All calling processes must have the correct *m* and *n*.
- Processes in context *A* must correctly define all parameters describing *A*.
- Processes in context *B* must correctly define all parameters describing *B*.
- Processes which are not members of context *A* must pass *ctxt_a* = -1 and need not set other parameters describing *A*.
- Processes which are not members of context*B* must pass *ctxt_b* = -1 and need not set other parameters describing *B*.

Because of its generality, p?gemr2d can be used for many operations not usually associated with copy routines. For instance, it can be used to a take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for instance, this routine can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this routine requires an array descriptor with *dtype_* = 1.

Input Parameters

m (global) INTEGER. The number of rows of matrix *A* to be copied (*m*≥0).

n (global) INTEGER. The number of columns of matrix *A* to be copied (*n*≥0).

a (local)

REAL for psgemr2d

DOUBLE for pdgemr2d

COMPLEX for pcgemr2d

DOUBLE COMPLEX for pzgemr2d

INTEGER for pigemr2d.

Pointer into the local memory to array of size *lld_a* by *LOCc*(*ja*+*n*-1) containing the source matrix *A*.

ia, *ja* (global) INTEGER. The row and column indices in the array *A* indicating the first row and the first column, respectively, of the submatrix of *A*) to copy. 1 ≤*ia*≤*total_rows_in_a* - *m* +1, 1 ≤*ja*≤*total_columns_in_a* - *n* +1.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Only *dtype_a* = 1 is supported, so *dlen_* = 9.

If the calling process is not part of the context of *A*, *ctxt_a* must be equal to -1.

ib, *jb* (global) INTEGER. The row and column indices in the array *B* indicating the first row and the first column, respectively, of the submatrix *B* to which to copy the matrix. 1 ≤*ib*≤*total_rows_in_b* - *m* +1, 1 ≤*jb*≤*total_columns_in_b* - *n* +1.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Only *dtype_b* = 1 is supported, so *dlen_* = 9.

If the calling process is not part of the context of *B*, *ctxt_b* must be equal to -1.

ictxt (global)INTEGER.

The context encompassing at least the union of all processes in context *A* and context *B*. All processes in the context *ictxt* must call this routine, even if they do not own a piece of either matrix.

Output Parameters

b

REAL for psgemr2d

DOUBLE for pdgemr2d

COMPLEX for pcgemr2d

DOUBLE COMPLEX for pzgemr2d

INTEGER for pigemr2d.

Pointer into the local memory to array of size *lld_b* by *LOCc(jb+n-1)*.

Overwritten by the submatrix from *A*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trmr2d

Copies a submatrix from one trapezoidal matrix to another.

Syntax

```
call pstrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pdtrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pctrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pztrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pitrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
```

Description

The p?trmr2d routine copies the indicated matrix or submatrix of *A* to the indicated matrix or submatrix of *B*. It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With p?gemr2d, these routines are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix *A* in context *A* (distributed over process grid *A*) and copy it to a matrix or submatrix *B* in context *B* (distributed over process grid *B*).

The p?trmr2d routine assumes the matrix or submatrix to be trapezoidal. Only the upper or lower part is copied, and the other part is unchanged.

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context *A* is disjoint from context *B*. The general rules for which parameters need to be set are:

- All calling processes must have the correct *m* and *n*.
- Processes in context *A* must correctly define all parameters describing *A*.
- Processes in context *B* must correctly define all parameters describing *B*.
- Processes which are not members of context *A* must pass *ctxt_a* = -1 and need not set other parameters describing *A*.
- Processes which are not members of context *B* must pass *ctxt_b* = -1 and need not set other parameters describing *B*.

Because of its generality, p?trmr2d can be used for many operations not usually associated with copy routines. For instance, it can be used to a take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for instance, this routine can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this routine requires an array descriptor with *dtype_* = 1.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

uplo (global) CHARACTER*1. Specifies whether to copy the upper or lower part of the matrix or submatrix.

uplo = 'U'	Copy the upper triangular part.
uplo = 'L'	Copy the lower triangular part.

diag (global) CHARACTER*1. Specifies whether to copy the diagonal of the matrix or submatrix.

diag = 'U'	Do not copy the diagonal.
diag = 'N'	Copy the diagonal.

m (global) INTEGER. The number of rows of matrix *A* to be copied ($m \geq 0$).

n (global) INTEGER. The number of columns of matrix *A* to be copied ($n \geq 0$).

a (local)

REAL for pstrmr2d

DOUBLE for pdtrmr2d

COMPLEX for pctrmr2d

DOUBLE COMPLEX for pztrmr2d

INTEGER for pitrmr2d.

Pointer into the local memory to array of size *lld_a* by *LOCc(ja+n-1)* containing the source matrix *A*.

ia, *ja* (global) INTEGER. The row and column indices in the array *A* indicating the first row and the first column, respectively, of the submatrix of *A*) to copy. $1 \leq ia \leq total\_rows\_in\_a - m + 1$, $1 \leq ja \leq total\_columns\_in\_a - n + 1$.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Only *dtype_a* = 1 is supported, so *dlen_* = 9.

If the calling process is not part of the context of *A*, *ctxt_a* must be equal to -1.

ib, *jb* (global) INTEGER. The row and column indices in the array *B* indicating the first row and the first column, respectively, of the submatrix *B* to which to copy the matrix. $1 \leq ib \leq total\_rows\_in\_b - m + 1$, $1 \leq jb \leq total\_columns\_in\_b - n + 1$.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Only *dtype_b* = 1 is supported, so *dlen_* = 9.

If the calling process is not part of the context of *B*, *ctxt_b* must be equal to -1.

ictxt (global)INTEGER.

The context encompassing at least the union of all processes in context *A* and context *B*. All processes in the context *ictxt* must call this routine, even if they do not own a piece of either matrix.

Output Parameters

b REAL for pstrmr2d

DOUBLE for pdtrmr2d

COMPLEX for pctrmr2d

DOUBLE COMPLEX for pztrmr2d

INTEGER for pitrmr2d.

Pointer into the local memory to array of size *lld_b* by *LOCc(jb+n-1)*.

Overwritten by the submatrix from *A*.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Sparse Solver Routines

Intel® oneAPI Math Kernel Library (oneMKL) sparse solver algorithms for solving real or complex, symmetric, structurally symmetric or nonsymmetric, positive definite, indefinite or Hermitian square sparse linear system of algebraic equations.

The terms and concepts required to understand the use of the Intel® oneAPI Math Kernel Library (oneMKL) sparse solver routines are discussed in the Appendix "Linear Solvers Basics". If you are familiar with linear sparse solvers and sparse matrix storage schemes, you can skip these sections and go directly to the interface descriptions.

See the description of

- the direct sparse solver based on PARDISO*, which is referred to here as Intel MKL PARDISO;
- the alternative interface for the direct sparse solver, which is referred to here as the DSS interface;
- iterative sparse solvers (ISS) based on the reverse communication interface (RCI);
- preconditioners based on the incomplete LU factorization technique.
- a direct sparse solver based on QR decomposition.

oneMKL PARDISO - Parallel Direct Sparse Solver Interface

This section describes the interface to the shared-memory multiprocessing parallel direct sparse solver known as the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver.

The Intel® oneAPI Math Kernel Library (oneMKL) PARDISO package is a high-performance, robust, memory efficient, and easy to use software package for solving large sparse linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques [Schenk00-2]. To improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is used with a combination of left- and right-looking supernode techniques [Schenk00, Schenk01, Schenk02, Schenk03]. The parallel pivoting

methods allow complete supernode pivoting to compromise numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

The following table lists the names of the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO routines and describes their general use.

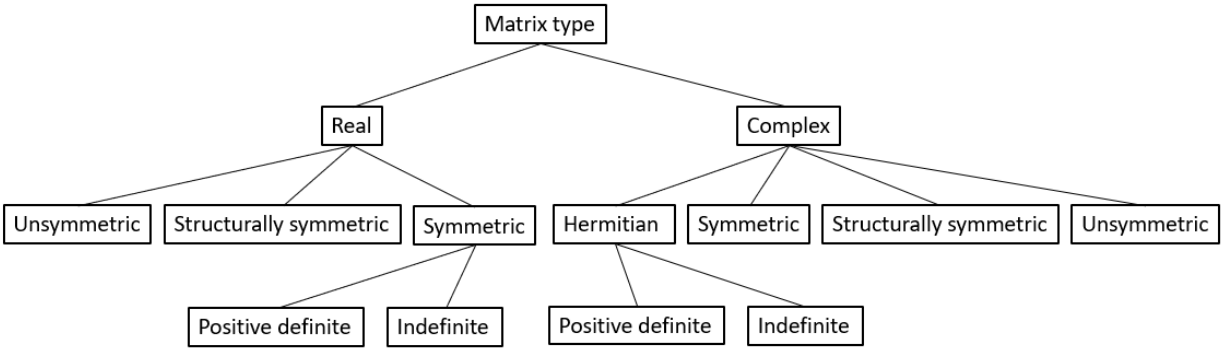
oneMKL PARDISO Routines

Routine	Description
pardisoinit	Initializes Intel® oneAPI Math Kernel Library (oneMKL) PARDISO with default parameters depending on the matrix type.
pardiso	Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.
pardiso_64	Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.
mkl_pardiso_pivot	Replaces routine which handles Intel® oneAPI Math Kernel Library (oneMKL) PARDISO pivots with user-defined routine.
pardiso_getdiag	Returns diagonal elements of initial and factorized matrix.
pardiso_export	Places pointers dedicated for sparse representation of requested matrix into MKL PARDISO.
pardiso_handle_store	Store internal structures from `pardiso` to a file.
pardiso_handle_restore	Restore `pardiso` internal structures from a file.
pardiso_handle_delete	Delete files with `pardiso` internal structure data.
pardiso_handle_store_64	Store internal structures from `pardiso_64` to a file.
pardiso_handle_restore_64	Restore `pardiso_64` internal structures from a file.
pardiso_handle_delete_64	Delete files with `pardiso_64` internal structure data.

The Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver supports a wide range of real and complex sparse matrix types (seethe figure below).

__border__top

Sparse Matrices That Can Be Solved with the oneMKL PARDISO Solver



The Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver performs four tasks:

- analysis and symbolic factorization
- numerical factorization
- forward and backward substitution including iterative refinement
- termination to release all internal solver memory.

To find code examples that use Intel® oneAPI Math Kernel Library (oneMKL) PARDISO routines to solve systems of linear equations, unzip theappropriate Fortran archive file in the `examples`folder of the Intel® oneAPI Math Kernel Library (oneMKL) installation directory. Code examples will be in the`examples/solverf/source` folder.

Supported Matrix Types

The analysis steps performed by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO depend on the structure of the input matrix*A*.

Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation *P* based on either the minimum degree algorithm [Liu85] or the nested dissection algorithm from the METIS package [Karypis98] (both included with Intel® oneAPI Math Kernel Library (oneMKL)), followed by the parallel left-right looking numerical Cholesky factorization [Schenk00-2] of $PAP^T = LL^T$ for symmetric positive-definite matrices, or $PAP^T = LDL^T$ for symmetric indefinite matrices. The solver uses diagonal pivoting, or 1x1 and 2x2 Bunch-Kaufman pivoting for symmetric indefinite matrices. An approximation of *X* is found by forward and backward substitution and optional iterative refinement.

Whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal supernode block, the coefficient matrix is perturbed. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricting notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. Furthermore, for a large set of matrices from different applications areas, this method is as accurate as a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Another method of improving the pivoting accuracy is to use symmetric weighted matching algorithms. These algorithms identify large entries in the coefficient matrix A that, if permuted close to the diagonal, permit the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. These algorithms are based on maximum weighted matchings and improve the quality of the factor in a complementary way to the alternative of using more complete pivoting techniques.

The inertia is also computed for real symmetric indefinite matrices.

Structurally Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation P followed by the parallel numerical factorization of $PAP^T = QLU^T$. The solver uses partial pivoting in the supernodes and an approximation of X is found by forward and backward substitution and optional iterative refinement.

Nonsymmetric Matrices

The solver first computes a nonsymmetric permutation P_{MPS} and scaling matrices D_r and D_c with the aim of placing large entries on the diagonal to enhance reliability of the numerical factorization process [Duff99]. In the next step the solver computes a fill-in reducing permutation P based on the matrix $P_{MPS}A + (P_{MPS}A)^T$ followed by the parallel numerical factorization

$QLUR = PP_{MPS}D_rAD_cP$

with supernode pivoting matrices Q and R. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99]. The magnitude of the potential pivot is tested against a constant threshold of

$\texttt{alpha} = \texttt{eps} * ||A2||_{\texttt{inf}}$,

where `eps` is the machine precision, $A2 = P*P_{MPS}*D_r*A*D_c*P$, and $||A2||_{\texttt{inf}}$ is the infinity norm of A. Any tiny pivots encountered during elimination are set to the sign $(l_{II})*\texttt{eps}*||A2||_{\texttt{inf}}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice the diagonal elements are rarely modified for a large class of matrices. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

Sparse Data Storage

Intel® oneAPI Math Kernel Library (oneMKL) PARDISO stores sparse data in several formats:

- CSR3: The 3-array variation of the compressed sparse row format described in Three Array Variation of CSR Format.
- BSR3: The three-array variation of the block compressed sparse row format described in Three Array Variation of BSR Format. Use `iparm(37)` to specify the block size.
- VBSR: Variable BSR format. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO analyzes the matrix provided in CSR3 format and converts it into an internal structure which can improve performance for matrices with a block structure. Use `iparm(37) = -t` ($0 < t \leq 100$) to specify use of internal VBSR format and to set the degree of similarity required to combine elements of the matrix. For example, if you set `iparm(37) = -80`, two rows of the input matrix are combined when their non-zero patterns are 80% or more similar.

NOTE
Intel® oneAPI Math Kernel Library (oneMKL) supports only the VBSR format for real and symmetric positive definite or indefinite matrices (*mtype* = 2 or *mtype* = -2).

Intel® oneAPI Math Kernel Library (oneMKL) supports these features for all matrix types as long as*iparm*(24)=1:

- *iparm*(31) > 0: Partial solution
- *iparm*(36) > 0: Schur complement
- *iparm*(60) > 0: OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO

For all storage formats, the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO parameter*ja* is used for the *columns* array, *ia* is used for *rowIndex*, and *a* is used for *values*. The algorithms in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO require column indices*ja* to be in increasing order per row and that the diagonal element in each row be present for any structurally symmetric matrix. For symmetric or nonsymmetric matrices the diagonal elements which are equal to zero are not necessary.

Caution
Intel® oneAPI Math Kernel Library (oneMKL) PARDISO column indices*ja* must be in increasing order per row. You can validate the sparse matrix structure with the matrix checker (*iparm*(27))

NOTE
While the presence of zero diagonal elements for symmetric matrices is not required, you should explicitly set zero diagonal elements for symmetric matrices. Otherwise, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO creates internal copies of arrays*ia*, *ja*, and *a* full of diagonal elements, which require additional memory and computational time. However, the memory and time required the diagonal elements in internal arrays is usually not significant compared to the memory and the time required to factor and solve the matrix.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Storage of Matrices

By default, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO stores data in RAM. This is referred to as In-Core (IC) mode. However, you can specify that Intel® oneAPI Math Kernel Library (oneMKL) PARDISO store matrices on disk by setting*iparm*(60). This mode is called the Out-of-Core (OOC) mode.

You can set the following parameters for the OOC mode.

Parameter/Environment Variable Name	Description
MKL_PARDISO_OOC_PATH	Directory for storing data created in the OOC mode.
MKL_PARDISO_OOC_FILE_NAME	Full file name (incl. path) which will be used for the OOC files

Parameter/Environment Variable Name	Description
MKL_PARDISO_OOC_MAX_CORE_SIZE	Maximum size of RAM (in megabytes) available for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO
MKL_PARDISO_OOC_MAX_SWAP_SIZE	Maximum swap size (in megabytes) available for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO
MKL_PARDISO_OOC_KEEP_FILE	A flag which determines whether temporary data files will be deleted or stored

By default, the current working directory is used in the OOC mode as a directory path for storing data. All work arrays will be stored in files named ooc_temp with different extensions. When MKL_PARDISO_OOC_FILE_NAME is not set and MKL_PARDISO_OOC_PATH is set, the names for the created files will contain <path>/mkl_pardiso or <path>\mkl_pardiso depending on the OS. Setting MKL_PARDISO_OOC_FILE_NAME=<filename> will override the path which could have been set in MKL_PARDISO_OOC_PATH. In this case <filename> will be used for naming the OOC files.

By default, MKL_PARDISO_OOC_MAX_CORE_SIZE is 2000 (MB) and MKL_PARDISO_OOC_MAX_SWAP_SIZE is 0.

NOTE
Do not set the sum of MKL_PARDISO_OOC_MAX_CORE_SIZE and MKL_PARDISO_OOC_MAX_SWAP_SIZE greater than the size of the RAM plus the size of the swap memory. Be sure to allow enough free memory for the operating system and any other processes which need to be running.

By default, all temporary data files will be deleted. For keeping them it is required to set MKL_PARDISO_OOC_KEEP_FILE to 0.

OOC parameters can be set in a configuration file. You can set the path to this file and its name using environmental variables MKL_PARDISO_OOC_CFG_PATH and MKL_PARDISO_OOC_CFG_FILE_NAME.

For setting parameters of OOC mode either environment variables or a configuration file can be used. When the last option is chosen, by default the name of the file is *pardiso_ooc.cfg* and it should be placed in the working directory. If needed, the user can set the path to the configuration file using environmental variables MKL_PARDISO_OOC_CFG_PATH and MKL_PARDISO_OOC_CFG_FILE_NAME. These variables specify the path and filename as follows:

- Linux* OS and OS X*: <MKL_PARDISO_OOC_CFG_PATH>/ <MKL_PARDISO_OOC_CFG_FILE_NAME>
- Windows* OS: <MKL_PARDISO_OOC_CFG_PATH>\<MKL_PARDISO_OOC_CFG_FILE_NAME>

An example of the configuration file:

```
MKL_PARDISO_OOC_PATH = <path>
MKL_PARDISO_OOC_MAX_CORE_SIZE = N
MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

Caution
The maximum length of the path lines in the configuration files is 1000 characters.

Alternatively, the OOC parameters can be set as environment variables via command line.

For Linux* OS and OS X*:

```
export MKL_PARDISO_OOC_PATH = <path>
export MKL_PARDISO_OOC_MAX_CORE_SIZE = N
export MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
export MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

For Windows* OS:

```
set MKL_PARDISO_OOC_PATH = <path>
set MKL_PARDISO_OOC_MAX_CORE_SIZE = N
set MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
set MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

where `<path>` should follow the OS naming convention.

Direct-Iterative Preconditioning for Nonsymmetric Linear Systems

The solver uses a combination of direct and iterative methods [Sonn89] to accelerate the linear solution process for transient simulation. Most applications of sparse solvers require solutions of systems with gradually changing values of the nonzero coefficient matrix, but with an identical sparsity pattern. In these applications, the analysis phase of the solvers has to be performed only once and the numerical factorizations are the important time-consuming steps during the simulation. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses a numerical factorization and applies the factors in a preconditioned Krylov Subspace iteration. If the iteration does not converge, the solver automatically switches back to the numerical factorization. This method can be applied to nonsymmetric matrices in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. You can select the method using the *iparm*(4) input parameter. The *iparm*(20) parameter returns the error status after running Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Single and Double Precision Computations

Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solves tasks using single or double precision. Each precision has its benefits and drawbacks. Double precision variables have more digits to store value, so the solver uses more memory for keeping data. But this mode solves matrices with better accuracy, which is especially important for input matrices with large condition numbers.

Single precision variables have fewer digits to store values, so the solver uses less memory than in the double precision mode. Additionally this mode usually takes less time. But as computations are made less precisely, only some systems of equations can be solved accurately enough using single precision.

Separate Forward and Backward Substitution

The solver execution step (see parameter *phase* = 33 below) can be divided into two or three separate substitutions: forward, backward, and possible diagonal. This separation can be explained by the examples of solving systems with different matrix types.

A real symmetric positive definite matrix *A* (*mtype* = 2) is factored by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO as $A = L*L^T$. In this case the solution of the system $A*x=b$ can be found as sequence of substitutions: $L*y=b$ (forward substitution, *phase* =331) and $L^T*x=y$ (backward substitution, *phase* =333).

A real nonsymmetric matrix *A* (*mtype* = 11) is factored by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO as $A = L*U$. In this case the solution of the system $A*x=b$ can be found by the following sequence: $L*y=b$ (forward substitution, *phase* =331) and $U*x=y$ (backward substitution, *phase* =333).

Solving a system with a real symmetric indefinite matrix *A* (*mtype* = -2) is slightly different from the cases above. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factors this matrix as $A=LDL^T$, and the solution of the system $A*x=b$ can be calculated as the following sequence of substitutions: $L*y=b$ (forward

substitution, *phase* =331), $D*y=y$ (diagonal substitution, *phase* =332), and finally $L^T*x=y$ (backward substitution, *phase* =333). Diagonal substitution makes sense only for symmetric indefinite matrices (*mtype* = -2, -4, 6). For matrices of other types a solution can be found as described in the first two examples.

Caution
The number of refinement steps (*iparm(8)*) must be set to zero if a solution is calculated with separate substitutions (*phase* = 331, 332, 333), otherwise Intel® oneAPI Math Kernel Library (oneMKL) PARDISO produces the wrong result.

NOTE
Different pivoting (*iparm(21)*) produces different LDL^T factorization. Therefore results of forward, diagonal and backward substitutions with diagonal pivoting can differ from results of the same steps with Bunch-Kaufman pivoting. Of course, the final results of sequential execution of forward, diagonal and backward substitution are equal to the results of the full solving step (*phase*=33) regardless of the pivoting used.

Callback Function for Pivoting Control

In-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO allows you to control pivoting with a callback routine,mkl_pardiso_pivot. You can then use the pardiso_getdiag routine to access the diagonal elements. Set iparm(56) to 1 in order to use the callback functionality.

Low Rank Update

Use low rank update to accelerate the factorization step in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO when you use multiple matrices with identical structure and similar values. After callingpardiso in the usual manner for factorization (*phase* = 12, 13, 22, or 23) for some matrix *A1*, low rank update can be applied to the factorization step (*phase* = 22 or 23) of some matrix *A2* with identical structure.

To use the low rank update feature, set *iparm*(39) = 1 while also setting *iparm*(24) = 10. Additionally, supply an array that lists the values in *A2* that are different from *A1* using the *perm* parameter as outlined in the pardiso *perm* parameter description.

Important
Low rank update can only be called for matrices with the exact same pattern of nonzero values. As such, the value of the *mtype*, *ia*, *ja*, and *iparm*(24) parameters should also be identical. In general, the low rank factorization should be called with the same parameters as the preceding factorization step for the same internal data structure handle (except for array *a*, *iparm*(39), and *perm*).

Low rank update does not currently support Intel TBB threading. In this case, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO defaults to full factorization instead.

Low rank update cannot be used in combination with a user-supplied permutation vector - in other words, you must use the default values of *iparm*(5) = 0, *iparm*(31) = 0, and *iparm*(36) = 0). Additionally, *iparm*(4), *iparm*(6), *iparm*(12), *iparm*(28), *iparm*(37), *iparm*(56), and *iparm*(60) must all be set to the default value of 0.

pardiso

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

`call pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs, iparm, msglvl, b, x, error)`

Include Files

- `mkl.fi, mkl_pardiso.f90`

Description

The `pardiso` routine calculates the solution of a set of sparse linear equations

```
A*X = B
```

with single or multiple right-hand sides, using a parallel *LU*, *LDL*, or LL^T factorization, where *A* is an *n*-by-*n* matrix, and *X* and *B* are *n*-by-*nrhs* vectors or matrices.

Notes

- This routine supports usage of the `mkl_progress` with OpenMP, TBB, and sequential threading. See `mkl_progress` for details. The case of *iparm*(24)=10 does not support this feature.
- If *iparm*(27) is set to 1 (Matrix checker), Intel® oneAPI Math Kernel Library PARDISO uses the auxiliary routine `sparse_matrix_checker` to check integer arrays `ia` and `ja`. `sparse_matrix_checker` has its own set of error values (from 21 to 24) that are returned in the event of an unsuccessful matrix check. For more details, refer to the sparse_matrix_checker documentation.

Input Parameters

NOTE
The types given for parameters in this section are specified in FORTRAN 77 notation. See Intel MKL PARDISO Parameters in Tabular Formfor detailed description of types of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO parameters in Fortran 90 notation.

pt

`INTEGER` for 32-bit or 64-bit architectures

`INTEGER*8` for 64-bit architectures

Array with size of 64.

Handle to internal data structure. The entries must be set to zero prior to the first call to `pardiso`. Unique for factorization.

Caution
After the first call to `pardiso` do not directly modify *pt*, as that could cause a serious memory leak.

Use the pardiso_handle_store or `pardiso_handle_store_64` routine to store the content of *pt* to a file. Restore the contents of *pt* from the file using pardiso_handle_restore or `pardiso_handle_restore_64`. Use `pardiso_handle_store` and `pardiso_handle_restore` with `pardiso`, and `pardiso_handle_store_64` and `pardiso_handle_restore_64` with `pardiso_64`.

maxfct INTEGER

Maximum number of factors with identical sparsity structure that must be kept in memory at the same time. In most applications this value is equal to 1. It is possible to store several different factorizations with the same nonzero structure at the same time in the internal data structure management of the solver.

`pardiso` can process several matrices with an identical matrix sparsity pattern and it can store the factors of these matrices at the same time. Matrices with a different sparsity structure can be kept in memory with different memory address pointers *pt*.

mnum INTEGER

Indicates the actual matrix for the solution phase. With this scalar you can define which matrix to factorize. The value must be: $1 \leq mnum \leq maxfct$.

In most applications this value is 1.

mtype INTEGER

Defines the matrix type, which influences the pivoting method. The Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver supports the following matrices:

1	real and structurally symmetric
2	real and symmetric positive definite
-2	real and symmetric indefinite
3	complex and structurally symmetric
4	complex and Hermitian positive definite
-4	complex and Hermitian indefinite
6	complex and symmetric
11	real and nonsymmetric
13	complex and nonsymmetric

phase INTEGER

Controls the execution of the solver. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including optional iterative refinement

  This phase can be divided into two or three separate substitutions: forward, backward, and diagonal (see Separate Forward and Backward Substitution).
- Memory release phase (*phase*= 0 or *phase*= -1)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

phase	**Solver Execution Steps**
11	Analysis
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
23	Numerical factorization, solve, iterative refinement
33	Solve, iterative refinement
331	like *phase*=33, but only forward substitution
332	like *phase*=33, but only diagonal substitution (if available)
333	like *phase*=33, but only backward substitution
0	Release internal memory for *L* and *U* matrix number *mnum*
-1	Release all internal memory for all matrices

If *iparm*(36) = 0, phases 331, 332, and 333 perform this decomposition:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{12} & L_{22} \end{bmatrix} \begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{21} \\ 0 & U_{22} \end{bmatrix}$$

If *iparm*(36) = 2, phases 331, 332, and 333 perform a different decomposition:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{12} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} U_{11} & U_{21} \\ 0 & I \end{bmatrix}$$

You can supply a custom implementation for phase 332 instead of calling pardiso. For example, it can be implemented with dense LAPACK functionality. Custom implementation also allows you to substitute the matrix *S* with your own.

NOTE
For very large Schur complement matrices use LAPACK functionality to compute the Schur complement vector instead of the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO phase 332 implementation.

n

INTEGER

Number of equations in the sparse linear systems of equations *A*X = B*. Constraint: *n* > 0.

a

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*(28) =0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*(28) =1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*(28) =0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*(28) =1)

Array. Contains the non-zero elements of the coefficient matrix *A* corresponding to the indices in *ja*. The coefficient matrix can be either real or complex. The matrix must be stored in the three-array variant of the compressed sparse row (CSR3) or in the three-array variant of the block compressed sparse row (BSR3) format, and the matrix must be stored with increasing values of *ja* for each row.

For CSR3 format, the size of *a* is the same as that of *ja*. Refer to the *values* array description in Three Array Variation of CSR Format for more details.

For BSR3 format the size of *a* is the size of *ja* multiplied by the square of the block size. Refer to the *values* array description in Three Array Variation of BSR Format for more details.

NOTE
If you set *iparm*(37) to a negative value, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO converts the data from CSR3 format to an internal variable BSR (VBSR) format. SeeSparse Data Storage.

ia

INTEGER

Array, size (*n*+1).

For CSR3 format, *ia*(*i*) (*i*≤*n*) points to the first column index of row *i* in the array *ja*. That is, *ia*(*i*) gives the index of the element in array *a* that contains the first non-zero element from row *i* of *A*. The last element *ia*(*n*+1) is taken to be equal to the number of non-zero elements in *A*, plus one. Refer to *rowIndex* array description in Three Array Variation of CSR Format for more details.

For BSR3 format, *ia*(*i*) (*i*≤*n*) points to the first column index of row *i* in the array *ja*. That is, *ia*(*i*) gives the index of the element in array *a* that contains the first non-zero block from row *i* of *A*. The last element *ia*(*n*+1) is taken to be equal to the number of non-zero blcoks in *A*, plus one. Refer to *rowIndex* array description in Three Array Variation of BSR Format for more details.

The array *ia* is accessed in all phases of the solution process.

Indexing of *ia* is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter *iparm*(35).

ja

INTEGER

For CSR3 format, array *ja* contains column indices of the sparse matrix *A*. It is important that the indices are in increasing order per row. For structurally symmetric matrices it is assumed that all diagonal elements are stored (even if they are zeros) in the list of non-zero elements in *a* and *ja*. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in Three Array Variation of CSR Format.

For BSR3 format, array *ja* contains column indices of the sparse matrix *A*. It is important that the indices are in increasing order per row. For structurally symmetric matrices it is assumed that all diagonal blocks are stored (even if they are zeros) in the list of non-zero blocks in *a* and *ja*. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in Three Array Variation of BSR Format.

The array *ja* is accessed in all phases of the solution process.

Indexing of *ja* is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter *iparm*(35).

perm

INTEGER

Array, size (*n*). Depending on the value of *iparm*(5) and *iparm*(31), holds the permutation vector of size *n*, specifies elements used for computing a partial solution, or specifies differing values of the input matrices for low rank update.

- If *iparm*(5) = 1, *iparm*(31) = 0, and *iparm*(36) = 0, *perm* specifies the fill-in reducing ordering to the solver. Let *A* be the original matrix and $C = P*A*P^T$ be the permuted matrix. Row (column) *i* of *C* is the *perm*(*i*) row (column) of *A*. The array *perm* is also used to return the permutation vector calculated during fill-in reducing ordering stage.

NOTE
Be aware that setting *iparm*(5) = 1 prevents use of a parallel algorithm for the solve step.

- If *iparm*(5) = 2, *iparm*(31) = 0, and *iparm*(36) = 0, the permutation vector computed in phase 11 is returned in the *perm* array.
- If *iparm*(5) = 0, *iparm*(31) > 0, and *iparm*(36) = 0, *perm* specifies elements of the right-hand side to use or of the solution to compute for a partial solution.
- If *iparm*(5) = 0, *iparm*(31) = 0, and *iparm*(36) > 0, *perm* specifies elements for a Schur complement.
- If *iparm*(39) = 1, *perm* specifies values that differ in *A* for low rank update (see Low Rank Update). The size of the array must be at least 2**ndiff* + 1, where *ndiff* is the number of values of *A* that are different. The values of *perm* should be:

  perm = {*ndiff*, *row_index1*, *column_index1*, *row_index2*, *column_index2*, ..., *row_index_ndiff*, *column_index_ndiff*}

  where *row_index_m* and *column_index_m* are the row and column indices of the *m*-th differing non-zero value in matrix *A*. The row and column index pairs can be in any order, but must use zero-based indexing regardless of the value of *iparm*(35).

See *iparm*(5), *iparm*(31), and *iparm*(39) for more details.

Indexing of *perm* is one-based by default, but unless *iparm*(39) = 1 it can be changed to zero-based by setting the appropriate value to the parameter *iparm*(35).

nrhs

INTEGER

Number of right-hand sides that need to be solved for.

iparm

INTEGER

Array, size (64). This array is used to pass various parameters to Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and to return some useful information after execution of the solver.

See pardiso iparm Parameter for more details about the *iparm* parameters.

msglvl

INTEGER

Message level information. If *msglvl* = 0 then pardiso generates no output, if *msglvl* = 1 the solver prints statistical information to the screen.

b

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*(28)=0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*(28)=1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*(28)=0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm(28)*=1)

Array, size (*n*, *nrhs*). On entry, contains the right-hand side vector/matrix *B*, which is placed in memory contiguously. The *b*(i+(k-1)×*nrhs*) element must hold the i-th component of k-th right-hand side vector. Note that *b* is only accessed in the solution phase.

Output Parameters

(See also Intel MKL PARDISO Parameters in Tabular Form.)

pt — Handle to internal data structure.

perm — See the Input Parameter description of the *perm* array.

iparm — On output, some *iparm* values report information such as the numbers of non-zero elements in the factors.

See pardiso iparm Parameter for more details about the *iparm* parameters.

b — On output, the array is replaced with the solution if *iparm(6)* = 1.

x

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm(28)*=0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm(28)*=1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm(28)*=0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm(28)*=1)

Array, size (*n*, *nrhs*). If *iparm(6)*=0 it contains solution vector/matrix *X*, which is placed contiguously in memory. The *x*(i+(k-1)× *n*) element must hold the i-th component of the k-th solution vector. Note that *x* is only accessed in the solution phase.

error — INTEGER

The error indicator according to the below table:

error	**Information**
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem

error	Information
-4	Zero pivot, numerical factorization or iterative refinement problem. If the error appears during the solution phase, try to change the pivoting perturbation (*iparm*(10)) and also increase the number of iterative refinement steps. If it does not help, consider changing the scaling, matching and pivoting options (*iparm*(11), *iparm*(13), *iparm*(21))
-5	unclassified (internal) error
-6	reordering failed (matrix types 11 and 13 only)
-7	diagonal matrix is singular
-8	32-bit integer overflow problem
-9	not enough memory for OOC
-10	error opening OOC files
-11	read/write error with OOC files
-12	(pardiso_64 only) pardiso_64 called from 32-bit library
-13	interrupted by the (user-defined) mkl_progress function
-15	internal error which can appear for *iparm*(24)=10 and *iparm*(13)=1. Try switch matching off (set *iparm*(13)=0 and rerun.)

pardisoinit

Initialize Intel® oneAPI Math Kernel Library (oneMKL) PARDISO with default parameters in accordance with the matrix type.

Syntax

```
call pardisoinit (pt, mtype, iparm)
```

Include Files

- mkl.fi, mkl_pardiso.f90

Description

This function initializes the solver handle *pt* for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO with zero values (as needed for the very first call of pardiso) and sets default *iparm* values in accordance with the matrix type *mtype*.

The recommended way is to avoid using pardisoinit and to initialize *pt* and set the values of the *iparm* array manually as the default parameters might not be the best for a particular use case.

An alternative method to set default *iparm* values is to call pardiso in the analysis phase with iparm(1)=0. In this case, the solver handle *pt* must be initialized with zero values.

The pardisoinit routine initializes only the in-core version of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. Switching to the out-of-core version of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO as well as changing default *iparm* values can be done after the call to pardisoinit but before the first call to pardiso.

The pardisoinit routine cannot be used together with the pardiso_64 routine.

Input Parameters

mtype

INTEGER

Matrix type. Based on this value pardisoinit chooses default values for the *iparm* array. Refer to the section oneMKL PARDISO Parameters in Tabular Formfor more details about the default values of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Output Parameters

pt

INTEGER for 32-bit or 64-bit architectures

INTEGER*8 for 64-bit architectures

Array of size 64. Handle to internal data structure. The pardisoinit routine nullifies the array *pt*.

NOTE
It is very important that *pt* is initialized with zero before the first call of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. After that first call you must never modify the array, because it could cause a serious memory leak or a crash.

iparm

INTEGER

Array of size 64. This array is used to set various options for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and to return some useful information after execution of the solver. Thepardisoinit routine fills in the *iparm* array with the default values. Refer to the section oneMKL PARDISO Parameters in Tabular Form for more details about the default values of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

pardiso_64

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.

Syntax

```
call pardiso_64 (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs, iparm, msglvl, b, x, error)
```

Include Files

- mkl.fi, mkl_pardiso.f90

Description

pardiso_64 is an alternative ILP64 (64-bit integer) version of the pardiso routine (see Description section for more details). The interface of pardiso_64 is the same as the interface of pardiso, but it accepts and returns all INTEGER data as INTEGER*8.

Use pardiso_64 when pardisofor solving large matrices (with the number of non-zero elements on the order of 500 million or more). You can use it together with the usual LP64 interfaces for the rest of Intel® oneAPI Math Kernel Library (oneMKL) functionality. In other words, if you use 64-bit integer version (pardiso_64), you do not need to re-link your applications with ILP64 libraries. Take into account that pardiso_64 may perform slower than regular pardiso on the reordering and symbolic factorization phase.

NOTE
pardiso_64 is supported only in the 64-bit libraries. If pardiso_64 is called from the 32-bit libraries, it returns *error* =-12.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

The input parameters of pardiso_64 are the same as the input parameters of pardiso, but pardiso_64 accepts all INTEGER data as INTEGER*8.

Output Parameters

The output parameters of pardiso_64 are the same as the output parameters of pardiso, but pardiso_64 returns all INTEGER data as INTEGER*8.

mkl_pardiso_pivot

Replaces routine which handles Intel® oneAPI Math Kernel Library (oneMKL) PARDISO pivots with user-defined routine.

Syntax

```
call mkl_pardiso_pivot (ai, bi, eps)
```

Include Files

- mkl.fi, mkl_pardiso.f90

Description

The mkl_pardiso_pivotroutine allows you to handle diagonal elements which arise during numerical factorization that are zero or near zero. By default, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO determines that a diagonal element*bi* is a pivot if *bi* < *eps*, and if so, replaces it with *eps*. But you can provide your own routine to modify the resulting factorized matrix in case there are small elements on the diagonal during the factorization step.

NOTE
To use this routine, you must set *iparm*(56) to 1 before the main pardiso loop.

NOTE
This routine is only available for in-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Input Parameters

ai — DOUBLE PRECISION - for real types of matrices (*mtype*=2, -2, 4, and 6) and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*(28) =0)

Diagonal element of initial matrix corresponding to pivot element.

bi — DOUBLE PRECISION - for real types of matrices (*mtype*=2, -2, 4, and 6) and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*(28) =0)

Diagonal element of factorized matrix that could be chosen as a pivot element.

eps — DOUBLE PRECISION

Scalar to compare with diagonal of factorized matrix. On input equal to parameter described by *iparm*(10).

Output Parameters

bi — In case element is chosen as a pivot, value with which to replace the pivot.

pardiso_getdiag

Returns diagonal elements of initial and factorized matrix.

Syntax

```
call pardiso_getdiag (pt, df, da, mnum, error)
```

Include Files

- mkl.fi, mkl_pardiso.f90

Description

This routine returns the diagonal elements of the initial and factorized matrix for a real or Hermitian matrix.

NOTE
In order to use this routine, you must set iparm(56) to 1 before the main pardiso loop.

If iparm(24) is set to 10 (an improved two-level factorization algorithm for nonsymmetric matrices), Intel® oneAPI Math Kernel Library PARDISO will automatically use the classic algorithm for factorization.

Input Parameters

pt — INTEGER for 32-bit or 64-bit architectures

INTEGER*8 for 64-bit architectures

Array with a size of 64. Handle to internal data structure for the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver. The entries must be set to zero prior to the first call to`pardiso`. Unique for factorization.

mnum

`INTEGER`

Indicates the actual matrix for the solution phase of the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver. With this scalar you can define the diagonal elements of the factorized matrix that you want to obtain. The value must be: 1 ≤*mnum* ≤ *maxfct*. In most applications this value is 1.

Output Parameters

df

`DOUBLE PRECISION`- for real types of matrices and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*`(28)` =0)

`REAL`- for real types of matrices and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*`(28)` =1)

`DOUBLE COMPLEX`- for complex types of matrices and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*`(28)` =0)

`COMPLEX`- for complex types of matrices and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*`(28)` =1)

Array with a dimension of *n*. Contains diagonal elements of the factorized matrix after factorization.

> **NOTE**
> Elements of *df* correspond to diagonal elements of matrix *L*computed during phase 22. Because during phase 22 Intel® oneAPI Math Kernel Library (oneMKL) PARDISO makes additional permutations to improve stability, it is possible that array*df* is not in line with the *perm* array computed during phase 11.

da

`DOUBLE PRECISION`- for real types of matrices and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*`(28)` =0)

`REAL`- for real types of matrices and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*`(28)` =1)

`DOUBLE COMPLEX`- for complex types of matrices and for double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*`(28)` =0)

`COMPLEX`- for complex types of matrices and for single precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (*iparm*`(28)` =1)

Array with a dimension of *n*. Contains diagonal elements of the initial matrix.

NOTE
Elements of *da* correspond to diagonal elements of matrix *L*computed during phase 22. Because during phase 22 Intel® oneAPI Math Kernel Library (oneMKL) PARDISO makes additional permutations to improve stability, it is possible that array*da* is not in line with the *perm* array computed during phase 11.

error INTEGER

The error indicator.

error	Information
0	no error
-1	Diagonal information not turned on before pardiso main loop (*iparm*(56)=0).

pardiso_export

Places pointers dedicated for sparse representation of a requested matrix (values, rows, and columns) into MKL PARDISO

Syntax

```
call pardiso_export(pt, values, rows, columns, step, iparm, error)
```

Include Files

- mkl.fi
- mkl_pardiso.f90

Description

This auxiliary routine places pointers dedicated for sparse representation of a requested matrix (values, rows, and columns) into MKL PARDISO. The matrix will be stored in the three-array variant of the compressed sparse row (CSR3 format) with 0-based indexing.

NOTE
Currently, this routine can be used only for a sparse Schur complement matrix. All parameters related to the Schur complement matrix (perm, iparm) must be set before the reordering stage of MKL PARDISO (phase = 11) is called.

Input Parameters

pt(64) INTEGER on 32-bit architectures; INTEGER*8 on 64-bit architectures

Array with a size of 64. Handle to internal data structure for the Intel® MKL PARDISO solver. The entries must be set to zero prior to the first call to pardiso. Unique for factorization.

iparm(64) INTEGER

This array is used to pass various parameters to Intel® MKL PARDISO and to return some useful information after execution of the solver.

step

INTEGER

Stage indicator. These are the currently supported values:

Step value	Notes
1	Used to place pointers related to a Schur complement matrix in MKL PARDISO. The routine with *step* equal to 1 must be called between the reordering and factorization phases of MKL PARDISO.
-1	Used to clean the internal handle.

Input/Output Parameters

values(*)

Parameter type: input/output parameter.

PARDISO_DATA_TYPE (see PARDISO_DATA_TYPE)

This array contains the non-zero elements of the requested matrix.

rows

Parameter type: input/output parameter.

INTEGER

For CSR3 format, `rows[i] ( i < size )` points to the first column index of row i in the array columns; that is, `rows[i]` gives the index of the element in the array values that contains the first non-zero element from row `i` of the sparse matrix. The last element, `rows[size]`, is equal to the number of non-zero elements in the sparse matrix.

columns

Parameter type: input/output parameter.

INTEGER

This array contains the column indices for the non-zero elements of the requested matrix.

error

Parameter type: output parameter.

INTEGER

The error status:

- `0` indicates no error.
- `1` indicates inconsistent input data.

Usage Example

The following C-style example demonstrates how to use the `pardiso_export` routine to get the sparse representation (that is, three-array CSR format) of a Schur complement matrix.

```
#include "mkl.h"

/*
 * Call the reordering phase of MKL PARDISO with iparm[35] set to -1 in
```

```
 * order to compute the Schur complement matrix only, or -2 to compute all
 * factorization arrays.  perm array indices related to the Schur complement
 * matrix must be set to 1.
 */
phase = 11;
for ( i = 0; i < schur_size; i++ ) { perm[i] = 1.; }
iparm[35] = -1;
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
  iparm, &msglvl, b, x, &error);

/*
 * After the reordering phase, iparm[35] will contain the number of non-zero
 * elements for the Schur complement matrix.  Arrays dedicated to the sparse
 * representation of the Schur complement matrix must be allocated before
 * the factorization stage of MKL PARDISO is called.
 */
schur_nnz     = iparm[35];
schur_rows    = (MKL_INT   *) mkl_malloc(schur_size+1, ALIGNMENT);
schur_columns = (MKL_INT   *) mkl_malloc(schur_nnz , ALIGNMENT);
schur_values  = (DATA_TYPE *) mkl_malloc(schur_nnz , ALIGNMENT);

/*
 * Call to the pardiso_export routine with step equal to 1 in order to put
 * pointers related to the three-array CSR format into MKL PARDISO:
 */
pardiso_export(pt, schur_values, schur_ia, schur_ja, &step, iparm, &error);

/*
 * Call the factorization phase of PARDISO with iparm[35] equal to -1 or -2
 * to compute the Schur complement matrix:
 */
phase = 22;
iparm[35] = -1;
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
  iparm, &msglvl, b, x, &error);

/*
 * After the factorization stage, schur_values, schur_rows, and
 * schur_columns will contain the Schur complement matrix in CSR3 format.
 */
```

pardiso_handle_store

Store internal structures from `pardiso` *to a file.*

Syntax

`call pardiso_handle_store (`*`pt`*`, `*`dirname`*`, `*`error`*`)`

Include Files

- `mkl.fi`, `mkl_pardiso.f90`

Description

This function stores Intel® oneAPI Math Kernel Library (oneMKL) PARDISO structures to a file, allowing you to store Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures between the stages of the`pardiso` routine. The pardiso_handle_restoreroutine can restore the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures from the file.

Input Parameters

pt

`INTEGER` for 32-bit or 64-bit architectures

`INTEGER*8` for 64-bit architectures

Array with a size of 64. Handle to internal data structure.

dirname

`CHARACTER`

String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string ("") to specify the current directory. The routine creates a file named `handle.pds` in the directory.

Output Parameters

pt

Handle to internal data structure.

error

`INTEGER`

The error indicator.

error	Information
`0`	No error.
`-2`	Not enough memory.
`-10`	Cannot open file for writing.
`-11`	Error while writing to file.
`-13`	Wrong file format.

pardiso_handle_restore

Restore `pardiso` *internal structures from a file.*

Syntax

`call pardiso_handle_restore (pt, dirname, error)`

Include Files

- `mkl.fi`, `mkl_pardiso.f90`

Description

This function restores Intel® oneAPI Math Kernel Library (oneMKL) PARDISO structures from a file. This allows you to restore Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures stored by`pardiso_handle_store` after a phase of the `pardiso` routine and continue execution of the next phase.

Input Parameters

dirname

`CHARACTER`

String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Output Parameters

pt

`INTEGER` for 32-bit or 64-bit architectures

`INTEGER*8` for 64-bit architectures

Array with a dimension of 64. Handle to internal data structure.

error

`INTEGER`

The error indicator.

error	Information
`0`	No error.
`-2`	Not enough memory.
`-10`	Cannot open file for reading.
`-11`	Error while reading from file.
`-13`	Wrong file format.

pardiso_handle_delete

Delete files with `pardiso` *internal structure data.*

Syntax

```
call pardiso_handle_delete (dirname, error)
```

Include Files

- `mkl.fi`, `mkl_pardiso.f90`

Description

This function deletes files generated with `pardiso_handle_store`that contain Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures.

Input Parameters

dirname

`CHARACTER`

String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Output Parameters

error

`INTEGER`

The error indicator.

error	Information
`0`	No error.
`-10`	Cannot delete files.

pardiso_handle_store_64

Store internal structures from `pardiso_64` *to a file.*

Syntax

`call pardiso_handle_store_64 (pt, dirname, error)`

Include Files

- `mkl.fi`, `mkl_pardiso.f90`

Description

This function stores Intel® oneAPI Math Kernel Library (oneMKL) PARDISO structures to a file, allowing you to store Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures between the stages of thepardiso_64 routine. The pardiso_handle_restore_64routine can restore the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures from the file.

Input Parameters

pt

`INTEGER*8` for 64-bit architectures

Array with a dimension of 64. Handle to internal data structure.

dirname

`CHARACTER`

String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string ("") to specify the current directory. The routine creates a file named `handle.pds` in the directory.

Output Parameters

pt

Handle to internal data structure.

error

`INTEGER`

The error indicator.

error	Information
`0`	No error.
`-2`	Not enough memory.
`-10`	Cannot open file for writing.
`-11`	Error while writing to file.
`-12`	Not supported in 32-bit library - routine is only supported in 64-bit libraries.
`-13`	Wrong file format.

pardiso_handle_restore_64

Restore `pardiso_64` *internal structures from a file.*

Syntax

`call pardiso_handle_restore_64 (pt, dirname, error)`

Include Files

- mkl.fi, mkl_pardiso.f90

Description

This function restores Intel® oneAPI Math Kernel Library (oneMKL) PARDISO structures from a file. This allows you to restore Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures stored bypardiso_handle_store_64 after a phase of the pardiso_64 routine and continue execution of the next phase.

Input Parameters

dirname CHARACTER

String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Input Parameters

pt INTEGER for 32-bit or 64-bit architectures

INTEGER*8 for 64-bit architectures

Array with a dimension of 64. Handle to internal data structure.

error INTEGER

The error indicator.

error	**Information**
0	No error.
-2	Not enough memory.
-10	Cannot open file for reading.
-11	Error while reading from file.
-13	Wrong file format.

pardiso_handle_delete_64

Syntax

Delete files with pardiso_64 internal structure data.

```
call pardiso_handle_delete_64 (dirname, error)
```

Include Files

- mkl.fi, mkl_pardiso.f90

Description

This function deletes files generated with pardiso_handle_store_64that contain Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures.

Input Parameters

dirname CHARACTER

String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Output Parameters

error INTEGER

The error indicator.

error	Information
0	No error.
-10	Cannot delete files.
-12	Not supported in 32-bit library - routine is only supported in 64-bit libraries.

oneMKL PARDISO Parameters in Tabular Form

The following table lists all parameters of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and gives their brief descriptions.

Parameter	Type	Description	Values	Comments	In/ Out
pt(64)	INTEGER on 32-bit architectures, INTEGER*8 on 64-bit architectures TYPE(MKL_PARDISO_HANDLE), INTENT(INOUT)	Solver internal data address pointer	0	Must be initialized with zeros and never be modified later	in/out
maxfct	INTEGER INTEGER, INTENT(IN)	Maximal number of factors in memory	>0	Generally used value is 1	in
mnum	INTEGER INTEGER, INTENT(IN)	The number of matrix (from 1 to *maxfct*) to solve	[1: *maxfct*]	Generally used value is 1	in
mtype	INTEGER INTEGER, INTENT(IN)	Matrix type	1 2 -2 3	Real and structurally symmetric Real and symmetric positive definite Real and symmetric indefinite Complex and structurally symmetric	in

Parameter	Type	Description	Values	Comments	In/ Out
			4	Complex and Hermitian positive definite	
			-4	Complex and Hermitian indefinite	
			6	Complex and symmetric matrix	
			11	Real and nonsymmetric matrix	
			13	Complex and nonsymmetric matrix	
phase	INTEGER INTEGER, INTENT(IN)	Controls the execution of the solver For *iparm*(36) > 0, phases 331, 332, and 333 perform a different decomposition. See the *phase* parameter of pardiso for details.	11	Analysis	in
			12	Analysis, numerical factorization	
			13	Analysis, numerical factorization, solve	
			22	Numerical factorization	
			23	Numerical factorization, solve	
			33	Solve, iterative refinement	
			331	*phase*=33, but only forward substitution	
			332	*phase*=33, but only diagonal substitution	
			333	*phase*=33, but only backward substitution	
			0	Release internal memory for L and U of the matrix number *mnum*	
			-1	Release all internal memory for all matrices	
n	INTEGER INTEGER, INTENT(IN)	Number of equations in the sparse linear system *A*X = B*	>0		in
a(*)	PARDISO_DATA_TYPE[1)] PARDISO_DATA_TYPE[1)], INTENT(IN)	Contains the non-zero elements of the coefficient matrix *A*	*	The size of *a* is the same as that of *ja*, and the coefficient matrix can be either real or complex. The matrix must be stored in the 3-array variation of compressed sparse row (CSR3) format with increasing values of *ja* for each row	in

Parameter	Type	Description	Values	Comments	In/ Out
ia(*n* + 1)	INTEGER INTEGER, INTENT(IN)	*rowIndex* array in CSR3 format	>=0	*ia*(*i*) gives the index of the element in array *a* that contains the first non-zero element from row *i* of *A*. The last element *ia*(*n*+1) (for one-based indexing) or *ia*(*n*) (for zero-based indexing) is taken to be equal to the number of non-zero elements in *A*, plus one. Note: *iparm*(35) indicates whether row/column indexing starts from 1 or 0.	in
ja(*)	INTEGER INTEGER, INTENT(IN)	*columns* array in CSR3 format	>=0	The column indices for each row of *A* must be sorted in increasing order. For structurally symmetric matrices zero diagonal elements must be stored in *a* and *ja*. Zero diagonal elements should be stored for symmetric matrices, although they are not required. For symmetric matrices, the solver needs only the upper triangular part of the system. Note: *iparm*(35) indicates whether row/column indexing starts from 1 or 0.	in
perm(*n*)	INTEGER INTEGER, INTENT(INOUT)	Holds the permutation vector of size *n*, specifies elements used for computing a partial solution, or specifies differing values of the input matrices for low rank update	>=0	You can apply your own fill-in reducing ordering (*iparm*(5) = 1) or return the permutation from the solver (*iparm*(5) = 2). Let $C = P*A*P^T$ be the permuted matrix. Row (column) *i* of *C* is the *perm*(*i*) row (column) of *A*. The numbering of the array must describe a permutation. To specify elements for a partial solution, set *iparm*(5) = 0, *iparm*(31) > 0, and *iparm*(36) = 0.	in/out

Parameter	Type	Description	Values	Comments	In/ Out
				To specify elements for a Schur complement, set *iparm*(5) = 0, *iparm*(31) = 0, and *iparm*(36) > 0. To specify values that differ in *A* for low rank update (see Low Rank Update), set *iparm*(39) = 1. The size of the array must be at least 2**ndiff* + 1, where *ndiff* is the number of values of *A* that are different. The values of *perm* should be: *perm* = {*ndiff*, *row_index1*, *column_index1*, *row_index2*, *column_index2*, ..., *row_index_ndiff*, *column_index_ndiff*} where *row_index_m* and *column_index_m* are the row and column indices of the *m*-th differing non-zero value in matrix *A*. The row and column index pairs can be in any order, but must use zero-based indexing regardless of the value of *iparm*(35). **NOTE** Unless you have specified low rank update, *iparm*(35) indicates whether row/column indexing starts from 1 or 0.	
nrhs	INTEGER INTEGER, INTENT(IN)	Number of right-hand sides that need to be solved for	>=0	Generally used value is 1 To obtain better Intel® oneAPI Math Kernel Library (oneMKL) PARDISO performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase.	in

Parameter	Type	Description	Values	Comments	In/ Out
iparm(64)	INTEGER INTEGER, INTENT(INOUT)	This array is used to pass various parameters to Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and to return some useful information after execution of the solver (see pardiso iparm Parameter for more details)	*	If *iparm*(1)=0, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO fills*iparm*(2) through *iparm*(64) with default values and uses them.	in/out
msglvl	INTEGER INTEGER, INTENT(IN)	Message level information	0 1	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO generates no output Intel® oneAPI Math Kernel Library (oneMKL) PARDISO prints statistical information	in
b(*n***nrhs*)	PARDISO_DATA_TYPE[1)] PARDISO_DATA_TYPE[1)], INTENT(INOUT)	Right-hand side vectors	*	On entry, contains the right-hand side vector/ matrix *B*, which is placed contiguously in memory. The *b*(i+(k-1)×*nrhs*) element must hold the *i*-th component of *k*-th right-hand side vector. Note that *b* is only accessed in the solution phase. On output, the array is replaced with the solution if *iparm*(6)=1.	in/out
x(*n***nrhs*)	PARDISO_DATA_TYPE[1)] PARDISO_DATA_TYPE[1)], INTENT(OUT)	Solution vectors	*	On output, if *iparm*(6)=0, contains solution vector/ matrix *X* which is placed contiguously in memory. The *x*(i+(k-1)×*nrhs*) element must hold the *i*-th component of *k*-th solution vector. Note that *x* is only accessed in the solution phase.	out
error	INTEGER INTEGER, INTENT(OUT)	Error indicator	0 -1 -2 -3	No error Input inconsistent Not enough memory Reordering problem	out

Parameter	Type	Description	Values	Comments	In/ Out
			-4	Zero pivot, numerical factorization or iterative refinement problem	
			-5	Unclassified (internal) error	
			-6	Reordering failed (matrix types 11 and 13 only)	
			-7	Diagonal matrix is singular	
			-8	32-bit integer overflow problem	
			-9	Not enough memory for OOC	
			-10	Problems with opening OOC temporary files	
			-11	Read/write problems with the OOC data file	

[1)] See description of `PARDISO_DATA_TYPE` in PARDISO_DATA_TYPE.

pardiso iparm Parameter

This table describes all individual components of the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO *iparm* parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

Component	Description
iparm(1) input	Use default values.
	0 — *iparm*(2) - *iparm*(64) are filled with default values.
	≠0 — You must supply all values in components *iparm*(2) - *iparm*(64).
iparm(2) input	Fill-in reducing ordering for the input matrix.

Caution
You can control the parallel execution of the solver by explicitly setting the `MKL_NUM_THREADS` environment variable. If fewer OpenMP threads are available than specified, the execution may slow down instead of speeding up. If `MKL_NUM_THREADS` is not defined, then the solver uses all available processors.

0	The minimum degree algorithm [Li99].
2*	The nested dissection algorithm from the METIS package [Karypis98].
3	The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel® oneAPI Math Kernel Library (oneMKL) PARDISO Phase 1 takes significant time.

NOTE
Setting *iparm*(2) = 3 prevents the use of CNR mode (*iparm*(34) > 0) because Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses dynamic parallelism.

Component	Description
iparm(3)	Reserved. Set to zero.
iparm(4) input	Preconditioned CGS/CG. This parameter controls preconditioned CGS [Sonn89] for nonsymmetric or structurally symmetric matrices and Conjugate-Gradients for symmetric matrices. *iparm*(4) has the form *iparm*(4) = 10**L*+*K*.

K=0	The factorization is always computed as required by *phase*.
K=1	CGS iteration replaces the computation of *LU*. The preconditioner is *LU* that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.
K=2	CGS iteration for symmetric positive definite matrices replaces the computation of LL^T. The preconditioner is LL^T that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.

The value *L* controls the stopping criterion of the Krylov Subspace iteration:

$\text{eps}_{CGS} = 10^{-L}$ is used in the stopping criterion

$||dx_i|| / ||dx_0|| < \text{eps}_{CGS}$

where $||dx_i|| = ||\text{inv}(L*U)*r_i||$ for $K = 1$ or $||dx_i|| = ||\text{inv}(L*L^T)*r_i||$ for $K = 2$ and r_i is the residue at iteration *i* of the preconditioned Krylov Subspace iteration.

A maximum number of 150 iterations is fixed with the assumption that the iteration will converge before consuming half the factorization time. Intermediate convergence rates and residue excursions are checked and can terminate the iteration process. If *phase* =23, then the factorization for a given *A* is automatically recomputed in cases where the Krylov Subspace iteration failed, and the corresponding direct solution is returned. Otherwise the solution from the preconditioned Krylov Subspace iteration is returned. Using *phase* =33 results in an error message (*error*=-4) if the stopping criteria for the Krylov Subspace iteration can not be reached. More information on the failure can be obtained from *iparm*(20).

The default is *iparm*(4)=0, and other values are only recommended for an advanced user. *iparm*(4) must be greater than or equal to zero.

Examples:

iparm(4)	Description
31	*LU*-preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices
61	*LU*-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices
62	LL^T-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric positive definite matrices

iparm(5) input	User permutation. This parameter controls whether user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms. Another use of this parameter is to control obtaining the fill-in reducing permutation vector calculated during the reordering stage of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Component	Description
	This option is useful for testing reordering algorithms, adapting the code to special applications problems (for instance, to move zero diagonal elements to the end of $P*A*P^T$), or for using the permutation vector more than once for matrices with identical sparsity structures. For definition of the permutation, see the description of the *perm* parameter. **Caution** You can only set one of *iparm*(5), *iparm*(31), and *iparm*(36), so be sure that the *iparm*(31) (partial solution) and the *iparm*(36) (Schur complement) parameters are 0 if you set *iparm*(5).
	0* User permutation in the *perm* array is ignored.
	1 Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the user supplied fill-in reducing permutation from the*perm* array. *iparm*(2) is ignored. **NOTE** Setting *iparm*(5) = 1 prevents use of a parallel algorithm for the solve step.
	2 Intel® oneAPI Math Kernel Library (oneMKL) PARDISO returns the permutation vector computed at phase 1 in the*perm* array.
iparm(6) input	Write solution on *x*. **NOTE** The array *x* is always used.
	0* The array *x* contains the solution; right-hand side vector *b* is kept unchanged.
	1 The solver stores the solution on the right-hand side *b*.
iparm(7) output	Number of iterative refinement steps performed. Reports the number of iterative refinement steps that were actually performed during the solve step.
iparm(8) input	Iterative refinement step. On entry to the solve and iterative refinement step, *iparm*(8) must be set to the maximum number of iterative refinement steps that the solver performs. **NOTE** Perturbed pivots result in iterative refinement (independent of the value of *iparm*(8)) and the number of executed iterations is reported in *iparm*(7).
	0* The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization.
	>0 Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of *iparm*(8) steps of iterative refinement. The solver might stop the process before the maximum number of steps if

Component	Description
	• a satisfactory level of accuracy of the solution in terms of backward error is achieved, • or if it determines that the required accuracy cannot be reached. In this case the solver returns -4 in the *error* parameter. The number of executed iterations is reported in *iparm*(7).
	<0 Maximum number of iterative refinement steps with a negative sign. Unlike the case above the accumulation of the residuum uses extended precision real and complex data types. **NOTE** Currently, this feature is only supported for sequential and OpenMP threading.
iparm(9) input	Tolerance level for the relative residual in the iterative refinement process. If set to a non-zero value, an additional criterion is used for stopping the iterative refinement: $\frac{\| r \|}{\| b \|} < 10^{-iparm(9)}$ If set to zero, default checks are used to determine when to stop the iterations (see iparm(8) description). **NOTE** Currently it is only used for iparm(24)=1 or 10 and OpenMP threading.
iparm(10) input	Pivoting perturbation. This parameter instructs Intel® oneAPI Math Kernel Library (oneMKL) PARDISO how to handle small pivots or zero pivots for nonsymmetric matrices (*mtype* =11 or *mtype* =13) and symmetric matrices (*mtype* =-2, *mtype* =-4, or *mtype* =6). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04]. Small pivots are perturbed with eps = $10^{-iparm(10)}$. The magnitude of the potential pivot is tested against a constant threshold of alpha = eps*$\|\|A2\|\|_{inf}$, where eps = $10^{(-iparm(10))}$, $A2 = P*P_{MPS}*D_r*A*D_c*P$, and $\|\|A2\|\|_{inf}$ is the infinity norm of the scaled and permuted matrix *A*. Any tiny pivots encountered during elimination are set to the sign (l_{II})*eps*$\|\|A2\|\|_{inf}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with eps = $10^{(-iparm(10))}$.
	13* The default value for nonsymmetric matrices(*mtype* =11, *mtype*=13), eps = 10^{-13}.
	8* The default value for symmetric indefinite matrices (*mtype* =-2, *mtype*=-4, *mtype*=6), eps = 10^{-8}.
iparm(11) input	Scaling vectors. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less than or equal to 1. This scaling method is applied only to nonsymmetric

Component	Description
	matrices (*mtype* = 11 or *mtype* = 13). The scaling can also be used for symmetric indefinite matrices (*mtype* = -2, *mtype* =-4, or *mtype* = 6) when the symmetric weighted matchings are applied (*iparm*(13) = 1). Use *iparm*(11) = 1 (scaling) and *iparm*(13) = 1 (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase (*phase*=11) you must provide the numerical values of the matrix *A* in array *a* in case of scaling and symmetric weighted matching.
	0* Disable scaling. Default for symmetric indefinite matrices.
	1* Enable scaling. Default for nonsymmetric matrices. Scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices (*mtype* = 11, *mtype* = 13). The scaling can also be used for symmetric indefinite matrices (*mtype* = -2, *mtype* = -4, *mtype* = 6) when the symmetric weighted matchings are applied (*iparm*(13) = 1). Note that in the analysis phase (*phase*=11) you must provide the numerical values of the matrix *A* in case of scaling.
iparm(12) input	Solve with transposed or conjugate transposed matrix *A*. **NOTE** For real matrices, the terms *transposed* and *conjugate transposed* are equivalent.
	0* Solve a linear system $AX = B$.
	1 Solve a conjugate transposed system $A^HX = B$ based on the factorization of the matrix *A*.
	2 Solve a transposed system $A^TX = B$ based on the factorization of the matrix *A*.
iparm(13) input	Improved accuracy using (non-) symmetric weighted matching. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization.
	0* Disable matching. Default for symmetric indefinite matrices.
	1* Enable matching. Default for nonsymmetric matrices. Maximum weighted matching algorithm to permute large elements close to the diagonal. It is recommended to use *iparm*(11) = 1 (scaling) and *iparm*(13) = 1 (matching) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems. Note that in the analysis phase (*phase*=11) you must provide the numerical values of the matrix *A* in case of symmetric weighted matching.
iparm(14) output	Number of perturbed pivots.

Component	Description	
	After factorization, contains the number of perturbed pivots for the matrix types: 1, 3, 11, 13, -2, -4 and 6.	
iparm(15) output	Peak memory on symbolic factorization. The total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase. This value is only computed in phase 1.	
iparm(16) output	Permanent memory on symbolic factorization. Permanent memory from the analysis and symbolic factorization phase in kilobytes that the solver needs in the factorization and solve phases. This value is only computed in phase 1.	
iparm(17) output	Size of factors/Peak memory on numerical factorization and solution. This parameter provides the size in kilobytes of the total memory consumed by in-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for internal floating point arrays. This parameter is computed in phase 1. See *iparm*(63) for the OOC mode. The total peak memory consumed by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO is max(*iparm*(15), *iparm*(16)+*iparm*(17))	
iparm(18) input/output	Report the number of non-zero elements in the factors.	
	<0	Enable reporting if *iparm*(18) < 0 on entry. The default value is -1.
	>=0	Disable reporting.
iparm(19) input/output	Report number of floating point operations (in 10^6 floating point operations) that are necessary to factor the matrix *A*.	
	<0	Enable report if *iparm*(19) < 0 on entry. This increases the reordering time.
	>=0 *	Disable report.
iparm(20) output	Report CG/CGS diagnostics.	
	>0	CGS succeeded, reports the number of completed iterations.
	<0	CG/CGS failed (*error*=-4 after the solution phase). If *phase*= 23, then the factors *L* and *U* are recomputed for the matrix *A* and the error flag *error*=0 in case of a successful factorization. If *phase* = 33, then *error* = -4 signals failure. *iparm*(20)= - it_cgs*10 - cgs_error. Possible values of cgs_error: 1 - fluctuations of the residuum are too large 2 - $\|\|dx_{max\_it\_cgs/2}\|\|$ is too large (slow convergence) 3 - stopping criterion is not reached at max_it_cgs 4 - perturbed pivots caused iterative refinement 5 - factorization is too fast for this matrix. It is better to use the factorization method with *iparm*(4) = 0
iparm(21) input	Pivoting for symmetric indefinite matrices.	

Component	Description
	NOTE Use *iparm*(11) = 1 (scaling) and *iparm*(13) = 1 (matchings) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.
	0 — Apply 1x1 diagonal pivoting during the factorization process.
	1* — Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of *mtype*=-2, *mtype*=-4, or *mtype*=6.
	2 — Apply 1x1 diagonal pivoting during the factorization process. Using this value is the same as using *iparm*(21) = 0 except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by *iparm*(8) (0 by default).
	3 — Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of *mtype*=-2, *mtype*=-4, or *mtype*=6. Using this value is the same as using *iparm*(21) = 1 except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by *iparm*(8) (0 by default).
iparm(22) output	Inertia: number of positive eigenvalues. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reports the number of positive eigenvalues for symmetric indefinite matrices.
iparm(23) output	Inertia: number of negative eigenvalues. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reports the number of negative eigenvalues for symmetric indefinite matrices.
iparm(24) input	Parallel factorization control.
	0* — Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the classic algorithm for factorization.
	1 — Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses a two-level factorization algorithm. This algorithm generally improves scalability in case of parallel factorization on many OpenMP threads (more than eight). **NOTE** Disable *iparm*(11) (scaling) and *iparm*(13) = 1 (matching) when using the two-level factorization algorithm. Otherwise Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the classic factorization algorithm.
	10 — Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses an improved two-level factorization algorithm for nonsymmetric matrices.
iparm(25) input	Parallel forward/backward solve control.
	0* — Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the following strategy for parallelizing the solving step: In the case of the one right-hand side, the parallelization will be performed by partitioning the matrix.

Component	Description	
		Otherwise, the parallelization will be over the right-hand sides. This feature is available only for in-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (see *iparm*(60)).
	1	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the sequential forward and backward solve.
	2	Independent from the number of the right-hand sides, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the parallel algorithm based on the matrix partitioning. This feature is available only for in-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (see *iparm*(60)).
iparm(26)	Reserved. Set to zero.	
iparm(27) input	Matrix checker.	
	0*	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO does not check the sparse matrix representation for errors.
	1	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO checks integer arrays *ia* and *ja*. In particular, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO checks whether column indices are sorted in increasing order within each row.
iparm(28) input	Single or double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. See *iparm*(8) for information on controlling the precision of the refinement steps. **Important** The *iparm*(28) value is stored in the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO handle between Intel® oneAPI Math Kernel Library (oneMKL) PARDISO calls, so the precision mode can be changed only during phase 1.	
	0*	Input arrays (*a*, *x* and *b*) and all internal arrays must be presented in double precision.
	1	Input arrays (*a*, *x* and *b*) must be presented in single precision. In this case all internal computations are performed in single precision.
iparm(29)	Reserved. Set to zero.	
iparm(30) output	Number of zero or negative pivots. If Intel® oneAPI Math Kernel Library (oneMKL) PARDISO detects zero or negative pivot for *mtype*=2 or *mtype*=4 matrix types, the factorization is stopped. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO returns immediately with an *error* = -4, and *iparm*(30) reports the number of the equation where the zero or negative pivot is detected. Note: The returned value can be different for the parallel and sequential version in case of several zero/negative pivots.	
iparm(31) input	Partial solve and computing selected components of the solution vectors. This parameter controls the solve step of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. It can be used if only a few components of the solution vectors are needed or if you want to reduce the computation cost at the solve step by utilizing the sparsity of the right-hand sides. To use this option the input permutation vector define *perm* so	

<table>
<tr><th>Component</th><th>Description</th></tr>
<tr><td></td><td>
that when perm(i) = 1 it means that either the i-th component in the right-hand sides is nonzero, or the i-th component in the solution vectors is computed, or both, depending on the value of iparm(31).

The permutation vector permmust be present in all phases of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO software. At the reordering step, the software overwrites the input vectorperm by a permutation vector used by the software at the factorization and solver step. If m is the number of components such that perm(i) = 1, then the last m components of the output vector perm are a set of the indices i satisfying the condition perm(i) = 1 on input.

NOTE
Turning on this option often increases the time used by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for factorization and reordering steps, but it can reduce the time required for the solver step.

Important
You can use this feature for both in-core and out-of-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO as long as iparm[23]=1. Otherwise, you cannot use partial solve for out-of-core mode and you will need to setiparm(60)=0 for in-core mode. Set the parameters iparm(8) (iterative refinement steps), iparm(4) (preconditioned CGS), iparm(5) (user permutation), and iparm(36) (Schur complement) to 0 as well.

<table>
<tr><td>0*</td><td>Disables this option.</td></tr>
<tr><td>1</td><td>it is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector perm is defined so that perm(i) = 1 means that the (i)-th component in the right-hand sides is nonzero. In this case Intel® oneAPI Math Kernel Library (oneMKL) PARDISO only uses the non-zero components of the right-hand side vectors and computes only corresponding components in the solution vectors. That means thei-th component in the solution vectors is only computed if perm(i) = 1.</td></tr>
<tr><td>2</td><td>It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector perm is defined so that perm(i) = 1 means that the i-th component in the right-hand sides is nonzero.

Unlike for iparm(31)=1, all components of the solution vector are computed for this setting and all components of the right-hand sides are used. Because all components are used, for iparm(31)=2 you must set the i-th component of the right-hand sides to zero explicitly if perm(i) is not equal to 1.</td></tr>
<tr><td>3</td><td>Selected components of the solution vectors are computed. The perm array is not related to the right-hand sides and it only indicates which components of the solution vectors should be computed. In this case perm(i) = 1 means that the i-th component in the solution vectors is computed.</td></tr>
</table>
</td></tr>
<tr><td>iparm(32) - iparm(33)</td><td>Reserved. Set to zero.</td></tr>
<tr><td>iparm(34)

input</td><td>Optimal number of OpenMP threads for conditional numerical reproducibility (CNR) mode.

Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reads the value ofiparm(34) during the analysis phase (phase 1), so you cannot change it later.</td></tr>
</table>

Component	Description
	Because Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses C random number generator facilities during the analysis phase (phase 1) you must take these precautions to get numerically reproducible results: • Do not alter the states of the random number generators. • Do not run multiple instances of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO in parallel in the analysis phase (phase 1). **NOTE** CNR is only available for the in-core version of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and the non-parallel version of the nested dissection algorithm. You must also: • set *iparm*(60) to 0 in order to use the in-core version, • not set *iparm*(2) to 3 in order to not use the parallel version of the nested dissection algorithm. Otherwise Intel® oneAPI Math Kernel Library (oneMKL) PARDISO does not produce numerically repeatable results even if CNR is enabled for Intel® oneAPI Math Kernel Library (oneMKL) using the functionality described in Support Functions for CNR.
	0* — CNR mode for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO is enabled only if it is enabled for Intel® oneAPI Math Kernel Library (oneMKL) using the functionality described in Support Functions for CNR and the in-core version is used. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO determines the optimal number of OpenMP threads automatically, and produces numerically reproducible results regardless of the number of threads.
	>0 — CNR mode is enabled for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO if in-core version is used and the optimal number of OpenMP threads for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO to rely on is defined by the value of *iparm*(34). You can use *iparm*(34) to enable CNR mode independent from other Intel® oneAPI Math Kernel Library (oneMKL) domains. To get the best performance, set *iparm*(34) to the actual number of hardware threads dedicated for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. Setting *iparm*(34) to fewer OpenMP threads than the maximum number of them in use reduces the scalability of the problem being solved. Setting *iparm*(34) to more threads than are available can reduce the performance of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.
iparm(35) input	One- or zero-based indexing of columns and rows. **NOTE** Schur complement may be inaccurate or incorrect if pivots are detected. Please, check the output of *iparm*(29) .
	0* — One-based indexing: columns and rows indexing in arrays *ia*, *ja*, and *perm* starts from 1 (Fortran-style indexing).
	1 — Zero-based indexing: columns and rows indexing in arrays *ia*, *ja*, and *perm* starts from 0 (C-style indexing).
iparm(36) input/output	Schur complement matrix computation control. To calculate this matrix, you must set the input permutation vector *perm* to a set of indexes such that when *perm*(*i*) = 1, the *i*-th element of the initial matrix is an element of the Schur matrix.

Component	Description
	Caution You can only set one of *iparm*(5), *iparm*(31), and *iparm*(36), so be sure that the *iparm*(5) (user permutation) and the *iparm*(31) (partial solution) parameters are 0 if you set *iparm*(36).
	0* — Do not compute Schur complement.
	1 — Compute Schur complement matrix as part of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factorization step and return it in the solution vector. **NOTE** This option only computes the Schur complement matrix, and does not calculate factorization arrays.
	2 — Compute Schur complement matrix as part of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factorization step and return it in the solution vector. Since this option calculates factorization arrays you can use it to launch partial or full solution of the entire problem after the factorization step.
	-1 — Same as *iparm*(36) equals 1, but the Schur complement matrix is provided in 3-array CSR sparse format. Use in combination with pardiso_export. After reordering stage of MKL PARDISO, *iparm*(36) contains number of nonzero elements for Schur complement matrix. Set it once again before calling the factorization phase. **NOTE** This option is available only when *iparm*(24) is not equal to 0.
	-2 — Same as *iparm*(36) equals 2, but the Schur complement matrix is provided in 3-array CSR sparse format. Use in combination with pardiso_export. After reordering stage of MKL PARDISO, *iparm*(36) contains number of nonzero elements for Schur complement matrix. Set it once again before calling the factorization phase. **NOTE** This option is available only when *iparm*(24) is not equal to 0.
iparm(37) input	Format for matrix storage.
	0* — Use CSR format (see Three Array Variation of CSR Format) for matrix storage.
	> 0 — Use BSR format (see Three Array Variation of BSR Format) for matrix storage with blocks of size *iparm*(37).

Component	Description
	NOTE Intel® oneAPI Math Kernel Library (oneMKL) does not support BSR format in these cases: • *iparm*(11) > 0: Scaling vectors • *iparm*(13) > 0: Weighted matching • *iparm*(31) > 0: Partial solution • *iparm*(36) > 0: Schur complement • *iparm*(56) > 0: Pivoting control • *iparm*(60) > 0: OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO
	< 0 — Convert supplied matrix to variable BSR (VBSR) format (see Sparse Data Storage) for matrix storage. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO analyzes the matrix provided in CSR3 format and converts it to an internal VBSR format. Set *iparm*(37) = −*t*, 0 < *t*≤ 100. **NOTE** Intel® oneAPI Math Kernel Library (oneMKL) supports only the VBSR format for real and symmetric positive definite or indefinite matrices (*mtype* = 2 or *mtype*= -2). Intel® oneAPI Math Kernel Library (oneMKL) does not support VBSR format in these cases: • *iparm*(11) > 0: Scaling vectors • *iparm*(13) > 0: Weighted matching • *iparm*(56) > 0: Pivoting control
	NOTEIntel® oneAPI Math Kernel Library (oneMKL) supports these features for all matrix types as long as *iparm*(24) =1: • *iparm*(31) > 0: Partial solution • *iparm*(36) > 0: Schur complement • *iparm*(60) > 0: OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO
iparm(38)	Reserved. Set to zero.
iparm(39)	Enable low rank update (see Low Rank Update) to accelerate factorization for multiple matrices with identical structure and similar values.
	0* — Do not use low rank update functionality.
	1 — Use low rank update functionality. You must also set *iparm*(24) = 10 and provide a list of changed values in the *perm* array. This option requires the default settings of *iparm*(4), *iparm*(5), *iparm*(6), *iparm*(12), *iparm*(28), *iparm*(31), *iparm*(36), *iparm*(37), *iparm*(56), and *iparm*(60) as well.
iparm(40) - *iparm*(42)	Reserved. Set to zero.
iparm(43)	Control parameter for the computation of the diagonal of inverse matrix.
	0* — Do not compute the diagonal of inverse matrix.

Component	Description	
	1	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO computes the diagonal of the inverse matrix during the factorization phase. This feature is only available with two-level factorization algorithm (*iparm*(24) = 1) and real symmetric matrices (*mtype* = 2 or *mtype* = -2). The diagonal is returned in the solution vector.
iparm(44) - *iparm*(55)	Reserved. Set to zero.	
iparm(56)	Diagonal and pivoting control.	
	0*	Internal function used to work with pivot and calculation of diagonal arrays turned off.
	1	You can use the mkl_pardiso_pivot callback routine to control pivot elements which appear during numerical factorization. Additionally, you can obtain the elements of initial matrix and factorized matrices after the pardiso factorization step diagonal using the pardiso_getdiagroutine. This parameter can be turned on only in the in-core version of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.
iparm(57) - *iparm*(59)	Reserved. Set to zero.	
iparm(60) input	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO mode. *iparm*(60) switches between in-core (IC) and out-of-core (OOC) Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. OOC can solve very large problems by holding the matrix factors in files on the disk, which requires a reduced amount of main memory compared to IC. Unless you are operating in sequential mode, you can switch between IC and OOC modes after the reordering phase. However, you can get better Intel® oneAPI Math Kernel Library (oneMKL) PARDISO performance by setting*iparm*(60) before the reordering phase. The amount of memory used in OOC mode depends on the number of OpenMP threads. **Warning** Do not increase the number of OpenMP threads used for cluster_sparse_solver between the first call and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.	
	0*	IC mode.
	1	IC mode is used if the total amount of RAM (in megabytes) needed for storing the matrix factors is less than sum of two values of the environment variables: MKL_PARDISO_OOC_MAX_CORE_SIZE (default value 2000 MB) and MKL_PARDISO_OOC_MAX_SWAP_SIZE (default value 0 MB); otherwise OOC mode is used. In this case amount of RAM used by OOC mode cannot exceed the value of MKL_PARDISO_OOC_MAX_CORE_SIZE. If the total peak memory needed for storing the local arrays is more than MKL_PARDISO_OOC_MAX_CORE_SIZE, increase MKL_PARDISO_OOC_MAX_CORE_SIZE if possible.

Component	Description
	NOTE Conditional numerical reproducibility (CNR) is not supported for this mode.
	2 OOC mode. The OOC mode can solve very large problems by holding the matrix factors in files on the disk. Hence the amount of RAM required by OOC mode is significantly reduced compared to IC mode. If the total peak memory needed for storing the local arrays is more than `MKL_PARDISO_OOC_MAX_CORE_SIZE`, increase `MKL_PARDISO_OOC_MAX_CORE_SIZE` if possible. To obtain better Intel® oneAPI Math Kernel Library (oneMKL) PARDISO performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase. Refer to How to use Intel® MKL OOC PARDISO and Storage of Matrices for more details about OOC.
iparm(61) - *iparm*(62)	Reserved. Set to zero.
iparm(63) output	Size of the minimum OOC memory for numerical factorization and solution. This parameter provides the size in kilobytes of the minimum memory required by OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for internal floating point arrays. This parameter is computed in phase 1. Total peak memory consumption of OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO can be estimated as max(*iparm*(15), *iparm*(16) + *iparm*(63)).
iparm(64)	Reserved. Set to zero.

NOTE
Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See Sparse Matrix Storage Formats for an example.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

PARDISO_DATA_TYPE

The following table lists the values of `PARDISO_DATA_TYPE` depending on the matrix types and values of the parameter *iparm*(28).

Data type value	Matrix type *mtype*	*iparm*(28)	comments
DOUBLE PRECISION	1, 2, -2, 11	0	Real matrices, do
REAL		1	Real matrices, sin
DOUBLE COMPLEX	3, 6, 13, 4, -4	0	Complex matrices precision
COMPLEX		1	Complex matrices precision

Parallel Direct Sparse Solver for Clusters Interface

The Parallel Direct Sparse Solver for Clusters Interface solves large linear systems of equations with sparse matrices on clusters. It is

- high performing
- robust
- memory efficient
- easy to use

A hybrid implementation combines Message Passing Interface (MPI) technology for data exchange between parallel tasks (processes) running on different nodes, and OpenMP* technology for parallelism inside each node of the cluster. This approach effectively uses modern hardware resources such as clusters consisting of nodes with multi-core processors. The solver code is optimized for the latest Intel processors, but also performs well on clusters consisting of non-Intel processors.

Code examples are available in the Intel® oneAPI Math Kernel Library (oneMKL) installation`examples` directory.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Parallel Direct Sparse Solver for Clusters Interface Algorithm

Parallel Direct Sparse Solver for Clusters Interface solves a set of sparse linear equations

$A*X = B$

with multiple right-hand sides using a distributed LU, LL^T , LDL^T or $LDL*$ factorization, where A is an n-by-n matrix, and X and B are n-by-$nrhs$ matrices.

The solution comprises four tasks:

- analysis and symbolic factorization;
- numerical factorization;
- forward and backward substitution including iterative refinement;
- termination to release all internal solver memory.

The solver first computes a symmetric fill-in reducing permutation P based on the nested dissection algorithm from the METIS package [Karypis98](included with Intel® oneAPI Math Kernel Library (oneMKL)), followed by the Cholesky or other type of factorization (depending on matrix type)[Schenk00-2] of PAP^T. The solver uses either diagonal pivoting, or 1x1 and 2x2 Bunch and Kaufman pivoting for symmetric indefinite or Hermitian matrices before finding an approximation of X by forward and backward substitution and iterative refinement.

The initial matrix *A* is perturbed whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal blocks. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricted notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. For a large set of matrices from different application areas, the accuracy of this method is comparable to a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Parallel Direct Sparse Solver for Clusters additionally improves the pivoting accuracy by applying symmetric weighted matching algorithms. These methods identify large entries in the coefficient matrix *A* that, if permuted close to the diagonal, enable the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. The methods are based on maximum weighted matching and improve the quality of the factor in a complementary way to the alternative idea of using more complete pivoting techniques.

Parallel Direct Sparse Solver for Clusters Interface Matrix Storage

The sparse data storage in the Parallel Direct Sparse Solver for Clusters Interface follows the scheme described in the Sparse Matrix Storage Formats section using the variable *ja* for *columns*, *ia* for *rowIndex*, and *a* for *values*. Column indices *ja* must be in increasing order per row.

When an input data structure is not accessed in a call, a NULL pointer or any valid address can be passed as a placeholder for that argument.

Algorithm Parallelization and Data Distribution

Intel® oneAPI Math Kernel Library (oneMKL) Parallel Direct Sparse Solver for Clusters enables parallel execution of the solution algorithm with efficient data distribution.

The master MPI process performs the symbolic factorization phase to represent matrix *A* as computational tree. Then matrix *A* is divided among all MPI processes in a one-dimensional manner. The same distribution is used for *L-factor* (the lower triangular matrix in Cholesky decomposition). Matrix *A* and all required internal data are broadcast to subordinate MPI processes. Each MPI process fills in its own parts of *L-factor* with initial values of the matrix *A*.

Parallel Direct Sparse Solver for Clusters Interface computes all independent parts of *L-factor* completely in parallel. When a block of the factor must be updated by other blocks, these updates are independently passed to a temporary array on each updating MPI process. It further gathers the result into an updated block using the `MPI_Reduce()` routine. The computations within an MPI process are dynamically divided among OpenMP threads using pipelining parallelism with a combination of left- and right-looking techniques similar to those of the PARDISO* software. Level 3 BLAS operations from Intel® oneAPI Math Kernel Library (oneMKL) ensure highly efficient performance of block-to-block update operations.

During forward/backward substitutions, respective Right Hand Side (RHS) parts are distributed among all MPI processes. All these processes participate in the computation of the solution. Finally, the solution is gathered on the master MPI process.

This approach demonstrates good scalability on clusters with Infiniband* technology. Another advantage of the approach is the effective distribution of *L-factor* among cluster nodes. This enables the solution of tasks with a much higher number of non-zero elements than it is possible with any Symmetric Multiprocessing (SMP) in-core direct solver.

The algorithm ensures that the memory required to keep internal data on each MPI process is decreased when the number of MPI processes in a run increases. However, the solver requires that matrix *A* and some other internal arrays completely fit into the memory of each MPI process.

To get the best performance, run one MPI process per physical node and set the number of OpenMP* threads per node equal to the number of physical cores on the node.

NOTE
Instead of calling `MPI_Init()`, initialize MPI with `MPI_Init_thread()` and set the MPI threading level to `MPI_THREAD_FUNNELED` or higher. For details, see the code examples in *<install_dir>*`/examples`.

cluster_sparse_solver

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

`call cluster_sparse_solver (`*`pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs, iparm, msglvl, b, x, comm, error`*`)`

Include Files

- Fortran: `mkl_cluster_sparse_solver.f77`
- Fortran 90: `mkl_cluster_sparse_solver.f90`

Description

The routine `cluster_sparse_solver` calculates the solution of a set of sparse linear equations

```
A*X = B
```

with single or multiple right-hand sides, using a parallel *LU*, *LDL*, or LL^T factorization, where *A* is an *n*-by-*n* matrix, and *X* and *B* are *n*-by-*nrhs* vectors or matrices.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

NOTE
Most of the input parameters (except for the *pt*, *phase*, and *comm* parameters and, for the distributed format, the *a*, *ia*, and *ja* arrays) must be set on the master MPI process only, and ignored on other processes. Other MPI processes get all required data from the master MPI process using the MPI communicator, *comm*.

pt

`INTEGER*8` for 64-bit architectures

Array of size 64.

Handle to internal data structure. The entries must be set to zero before the first call to `cluster_sparse_solver`.

Caution
After the first call to `cluster_sparse_solver` do not modify *pt*, as that could cause a serious memory leak.

maxfct

`INTEGER`

Ignored; assumed equal to 1.

mnum INTEGER

Ignored; assumed equal to 1.

mtype INTEGER

Defines the matrix type, which influences the pivoting method. The Parallel Direct Sparse Solver for Clusters solver supports the following matrices:

1	real and structurally symmetric
2	real and symmetric positive definite
-2	real and symmetric indefinite
3	complex and structurally symmetric
4	complex and Hermitian positive definite
-4	complex and Hermitian indefinite
6	complex and symmetric
11	real and nonsymmetric
13	complex and nonsymmetric

phase INTEGER

Controls the execution of the solver. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. Parallel Direct Sparse Solver for Clusters has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including optional iterative refinement
- Memory release (*phase*= -1)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

phase	**Solver Execution Steps**
11	Analysis
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
23	Numerical factorization, solve, iterative refinement
33	Solve, iterative refinement
-1	Release all internal memory for all matrices

n

INTEGER

Number of equations in the sparse linear systems of equations *A*X = B*. Constraint: *n* > 0.

a

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28) =0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28) =1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28) =0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28) =1)

Array. Contains the non-zero elements of the coefficient matrix *A* corresponding to the indices in *ja*. The coefficient matrix can be either real or complex. The matrix must be stored in the three-array variant of the compressed sparse row (CSR3) or in the three-array variant of the block compressed sparse row (BSR3) format, and the matrix must be stored with increasing values of *ja* for each row.

For CSR3 format, the size of *a* is the same as that of *ja*. Refer to the *values* array description in Three Array Variation of CSR Format for more details.

For BSR3 format the size of *a* is the size of *ja* multiplied by the square of the block size. Refer to the *values* array description in Three Array Variation of BSR Format for more details.

NOTE
For centralized input (*iparm*(40)=0), provide the *a* array for the master MPI process only. For distributed assembled input (*iparm*(40)=1 or *iparm*(40)=2), provide it for all MPI processes.

Important
The column indices of non-zero elements of each row of the matrix *A* must be stored in increasing order.

ia

INTEGER

For CSR3 format, *ia*(*i*) (*i*≤*n*) points to the first column index of row *i* in the array *ja*. That is, *ia*(*i*) gives the index of the element in array *a* that contains the first non-zero element from row *i* of *A*. The last element *ia*(*n* +1) is taken to be equal to the number of non-zero elements in *A*, plus one. Refer to *rowIndex* array description in Three Array Variation of CSR Format for more details.

For BSR3 format, `ia(i)` (`i≤n`) points to the first column index of row *i* in the array `ja`. That is, `ia(i)` gives the index of the element in array `a` that contains the first non-zero block from row *i* of *A*. The last element `ia(n+1)` is taken to be equal to the number of non-zero blcoks in *A*, plus one. Refer to *rowIndex* array description in Three Array Variation of BSR Format for more details.

The array `ia` is accessed in all phases of the solution process.

Indexing of `ia` is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter `iparm(35)`. For zero-based indexing, the last element `ia(n+1)` is assumed to be equal to the number of non-zero elements in matrix *A*.

NOTE
For centralized input (`iparm(40)=0`), provide the `ia` array at the master MPI process only. For distributed assembled input (`iparm(40)=1` or `iparm(40)=2`), provide it at all MPI processes.

ja

INTEGER

For CSR3 format, array `ja` contains column indices of the sparse matrix *A*. It is important that the indices are in increasing order per row. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in Three Array Variation of CSR Format.

For BSR3 format, array `ja` contains column indices of the sparse matrix *A*. It is important that the indices are in increasing order per row. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in Three Array Variation of BSR Format.

The array `ja` is accessed in all phases of the solution process.

Indexing of `ja` is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter `iparm(35)`.

NOTE
For centralized input (`iparm(40)=0`), provide the `ja` array at the master MPI process only. For distributed assembled input (`iparm(40)=1` or `iparm(40)=2`), provide it at all MPI processes.

perm

INTEGER

Ignored.

nrhs

INTEGER

Number of right-hand sides that need to be solved for.

iparm

INTEGER

Array, size `64`. This array is used to pass various parameters to Parallel Direct Sparse Solver for Clusters Interface and to return some useful information after execution of the solver.

See cluster_sparse_solver iparm Parameter for more details about the *iparm* parameters.

msglvl

INTEGER

Message level information. If *msglvl* = 0 then cluster_sparse_solver generates no output, if *msglvl* = 1 the solver prints statistical information to the screen.

Statistics include information such as the number of non-zero elements in *L-factor* and the timing for each phase.

Set *msglvl* = 1 if you report a problem with the solver, since the additional information provided can facilitate a solution.

b

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Parallel Direct Sparse Solver for Clusters (*iparm*(28)=0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Parallel Direct Sparse Solver for Clusters (*iparm*(28)=1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Parallel Direct Sparse Solver for Clusters (*iparm*(28)=0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Parallel Direct Sparse Solver for Clusters (*iparm*(28)=1)

Array, size (*n*, *nrhs*). On entry, contains the right-hand side vector/matrix *B*, which is placed in memory contiguously. The *b*(i+(k-1)×*nrhs*) must hold the i-th component of k-th right-hand side vector. Note that *b* is only accessed in the solution phase.

comm

INTEGER

MPI communicator. The solver uses the Fortran MPI communicator internally.

Output Parameters

pt

Handle to internal data structure.

perm

Ignored.

iparm

On output, some *iparm* values report information such as the numbers of non-zero elements in the factors.

See cluster_sparse_solver iparm Parameter for more details about the *iparm* parameters.

b

On output, the array is replaced with the solution if *iparm*(6) = 1.

x

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Parallel Direct Sparse Solver for Clusters (*iparm*(28)=0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Parallel Direct Sparse Solver for Clusters (*iparm*(28)=1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Parallel Direct Sparse Solver for Clusters (*iparm*(28)=0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Parallel Direct Sparse Solver for Clusters (*iparm*(28)=1)

Array, size (*n*, *nrhs*). If *iparm*(6)=0 it contains solution vector/matrix *X*, which is placed contiguously in memory. The *x*(i+(k-1)× *n*) element must hold the i-th component of the k-th solution vector. Note that *x* is only accessed in the solution phase.

error

INTEGER

The error indicator according to the below table:

error	**Information**
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem
-4	Zero pivot, numerical factorization or iterative refinement problem. If the error appears during the solution phase, try to change the pivoting perturbation (*iparm*(10)) and also increase the number of iterative refinement steps. If it does not help, consider changing the scaling, matching and pivoting options (*iparm*(11), *iparm*(13), *iparm*(21))
-5	unclassified (internal) error
-6	reordering failed (matrix types 11 and 13 only)
-7	diagonal matrix is singular
-8	32-bit integer overflow problem
-9	not enough memory for OOC
-10	error opening OOC files
-11	read/write error with OOC files

cluster_sparse_solver_64

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

call cluster_sparse_solver_64 (*pt*, *maxfct*, *mnum*, *mtype*, *phase*, *n*, *a*, *ia*, *ja*, *perm*, *nrhs*, *iparm*, *msglvl*, *b*, *x*, *comm*, *error*)

Include Files

- Fortran: `mkl_cluster_sparse_solver.f77`
- Fortran 90: `mkl_cluster_sparse_solver.f90`

Description

The routine `cluster_sparse_solver_64` is an alternative ILP64 (64-bit integer) version of the cluster_sparse_solver routine (see the Description section for more details). The interface of `cluster_sparse_solver_64` is the same as the interface of `cluster_sparse_solver`, but it accepts and returns all INTEGER data as `INTEGER*8`.

Use `cluster_sparse_solver_64` when `cluster_sparse_solver`for solving large matrices (with the number of non-zero elements on the order of 500 million or more). You can use it together with the usual LP64 interfaces for the rest of Intel® oneAPI Math Kernel Library (oneMKL) functionality. In other words, if you use 64-bit integer version (`cluster_sparse_solver_64`), you do not need to re-link your applications with ILP64 libraries. Take into account that `cluster_sparse_solver_64` may perform slower than regular `cluster_sparse_solver` on the reordering and symbolic factorization phase.

NOTE
`cluster_sparse_solver_64` is supported only in the 64-bit libraries. If `cluster_sparse_solver_64` is called from the 32-bit libraries, it returns *error* =-12.

NOTE
This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

The input parameters of `cluster_sparse_solver_64` are the same as the input parameters of `cluster_sparse_solver`, but `cluster_sparse_solver_64` accepts all INTEGER data as `INTEGER*8`.

Output Parameters

The output parameters of `cluster_sparse_solver_64` are the same as the output parameters of `cluster_sparse_solver`, but `cluster_sparse_solver_64` returns all INTEGER data as `INTEGER*8`.

cluster_sparse_solver_get_csr_size

Computes the (local) number of rows and (local) number of nonzero entries for (distributed) CSR data corresponding to the provided name.

Syntax

`call cluster_sparse_solver_get_csr_size (pt, name, local_nrows, local_nnz, comm, error)`

Include Files

- `mkl_cluster_sparse_solver.f90`

Description

This routine uses the internal data created during the factorization phase of `cluster_sparse_solver` for matrix *A*. The routine then:

- Computes the local number of rows and the local number of nonzeros for CSR data that correspond to the provided *name*
- Returns the computed values in *local_nrows* and *local_nnz*

It is assumed that the CSR data defined by the *name* will be distributed in the same way as the matrix *A* (as defined by *iparm*(40)) used in cluster_sparse_solver.

The returned values can be used for allocating CSR arrays for factors *L* and *U*, and also for allocating arrays for permutations *P* and *Q*, or scaling matrix *D* which can then be used with cluster_sparse_solver_set_csr_ptrs or cluster_sparse_solver_set_ptr for exporting corresponding data via cluster_sparse_solver_export.

NOTE
Only call this routine after the factorization phase (*phase*=22) of the cluster_sparse_solver has been called. Neither *pt*, nor *iparm* should be changed after the preceding call to cluster_sparse_solver.

Input Parameters

pt

INTEGER*8 for 64-bit architectures

Array with size of 64.

Handle to internal data structure used in the prior calls to cluster_sparse_solver.

Caution
Do not modify *pt* after the calls to cluster_sparse_solver.

name

INTEGER

Specifies CSR data for which the output values are computed.

SPARSE_PTLUQT_L	Factor *L* from $P*A*Q=L*U$.
SPARSE_PTLUQT_U	Factor *U* from $P*A*Q=L*U$.
SPARSE_DPTLUQT_L	Factor *L* from $P*\ (D^{-1}A)*Q=L*U$.
SPARSE_DPTLUQT_U	Factor *U* from $P*\ (D^{-1}A)*Q=L*U$.

comm

INTEGER*4

MPI communicator. The solver uses the Fortran MPI communicator internally.

Output Parameters

local_nrows

INTEGER

The local number of rows for the CSR data which correspond to the *name*.

local_nnz

INTEGER

The local number of nonzero entries for the CSR data which correspond to the *name*.

error INTEGER

The error indicator:

error	Information
0	no error
-1	*pt* is a null pointer
-2	invalid *pt*
-3	invalid *name*
-4	unsupported *name*
-9	unsupported internal code path, consider switching off non-default *iparm* parameters for cluster_sparse_solver
-10	unsupported case when the matrix *A* is distributed among processes with overlap in the preceding calls to cluster_sparse_solver
-12	internal memory error

NOTE Refer to cl_solver_export_f90.f90 for an example using this functionality.

cluster_sparse_solver_set_csr_ptrs

Saves internally-provided pointers to the 3-array CSR data corresponding to the specified name.

Syntax

```
call cluster_sparse_solver_set_csr_ptrs (pt, name, rowptr, colindx, vals, comm, error)
```

Include Files

- mkl_cluster_sparse_solver.f90

Description

This routine internally saves the input pointers, *rowptr*, *colindx*, and *vals*, of the 3-array CSR data, which correspond to the provided *name*. It is assumed that the exported data will be distributed in the same way as the matrix *A* (as defined by *iparm*(40)) used in cluster_sparse_solver.

The saved pointers can then be used for exporting corresponding data by means of cluster_sparse_solver_export.

NOTE
Only call this routine after the factorization phase (*phase*=22) of the cluster_sparse_solver has been called. Neither *pt*, nor *iparm* should be changed after the preceding call to cluster_sparse_solver.

Input Parameters

pt

`INTEGER*8` for 64-bit architectures

Array with size of 64.

Handle to internal data structure used in the prior calls to `cluster_sparse_solver`.

Caution
Do not modify *pt* after the calls to `cluster_sparse_solver`.

name

`INTEGER`

Specifies for which CSR data the pointers are provided.

`SPARSE_PTLUQT_L`	Factor *L* from *P*A*Q=L*U*.
`SPARSE_PTLUQT_U`	Factor *U* from *P*A*Q=L*U*.
`SPARSE_DPTLUQT_L`	Factor *L* from *P* $(D^{-1}A)$*Q=L*U*.
`SPARSE_DPTLUQT_U`	Factor *U* from *P* $(D^{-1}A)$*Q=L*U*.

rowptr

`INTEGER`

Array of length at least (*local_nrows*+1) where *local_nrows* is the local number of rows, which can be obtained by calling `cluster_sparse_solver_get_csr_size`. This array contains row indices, such that *rowptr[i]* - *indexing* is the first index of row *i* in the array's *vals* and *colindx*. Here, the value of *indexing* is 0 for zero-based indexing and 1 for one-based indexing, and must be the same as it was for the matrix *A* used in the preceding calls to `cluster_sparse_solver` (also stored in *iparm(35)*).

Refer to *pointerB* array description in CSR Format for more details.

colindx

`INTEGER`

Array of length at least *rowptr[local_nrows]* – *rowptr[0]*. Indexing (zero- or one-based) must be the same as for *rowptr*. For one-based indexing, the array contains the column indices plus one for each non-zero element of the matrix which corresponds to the *name*. For zero-based indexing, the array contains the column indices for each non-zero element of the matrix.

vals

Array containing non-zero elements of the matrix which corresponds to the *name*. Its length is equal to length of the *colindx* array. Refer to values array description in CSR Format for more details.

It is interpreted internally as one of the following depending on `mtype` (type of the matrix *A*) and *iparm(28)* (precision) specified in the preceding call to `cluster_sparse_solver`.

`DOUBLE PRECISION`	For real types of matrices (`mtype`=1, 2, -2, and 11) and for double precision (*iparm(28)* = 0).

	REAL	For real types of matrices (mtype=1, 2, -2, and 11) and for single precision (*iparm(28)* = 1).
	DOUBLE COMPLEX	For complex types of matrices (mtype=3, 6, 13, 14, and -4) and for double precision (*iparm(28)* = 0).
	COMPLEX	For complex types of matrices (mtype=3, 6, 13, 14, and -4) and for single precision (*iparm(28)* = 1).

comm INTEGER*4

MPI communicator. The solver uses the Fortran MPI communicator internally.

Output Parameters

error INTEGER

The error indicator:

error	**Information**
0	no error
-1	*pt* is a null pointer
-2	invalid *pt*
-3	invalid *name*
-4	unsupported *name*
-9	unsupported internal code path, consider switching off non-default *iparm* parameters for cluster_sparse_solver
-12	internal memory error

NOTE Refer to cl_solver_export_f90.f90 for an example using this functionality.

cluster_sparse_solver_set_ptr

Internally saves a provided pointer to the data corresponding to the specified name.

Syntax

```
call cluster_sparse_solver_set_ptr (pt, name, ptr, comm, error)
```

Include Files

- mkl_cluster_sparse_solver.f90

Description

This routine internally saves the input pointer, *ptr*, of the data which correspond to the provided *name*. The saved pointer can then be used for exporting corresponding data by means of cluster_sparse_solver_export.

NOTE
Only call this routine after the factorization phase (*phase*=22) of the `cluster_sparse_solver` has been called. Neither *pt*, nor *iparm* should be changed after the preceding call to `cluster_sparse_solver`.

Input Parameters

pt

`INTEGER*8` for 64-bit architectures

Array with size of 64.

Handle to internal data structure used in the prior calls to `cluster_sparse_solver`.

Caution
Do not modify *pt* after the calls to `cluster_sparse_solver`.

name

`INTEGER`

Specifies the data for which the pointer is computed.

`SPARSE_PTLUQT_P`	Permutation *P* from $P*A*Q=L*U$.
`SPARSE_PTLUQT_Q`	Permutation *Q* from $P*A*Q=L*U$.
`SPARSE_DPTLUQT_P`	Permutation *P* from $P*(D^{-1}A)*Q=L*U$.
`SPARSE_DPTLUQT_Q`	Permutation *Q* from $P*(D^{-1}A)*Q=L*U$.
`SPARSE_DPTLUQT_D`	Scaling (diagonal) *D* from $P*(D^{-1}A)*Q=L*U$.

vals

Array containing elements of the vector representation for the data which corresponds to the *name*. Its length should be at least *local_nrows*, where *local_nrows* is the local number of rows in a corresponding matrix (obtained from `cluster_sparse_solver_get_csr_size`, for example).

For permutations *P* and *Q*, it is interpreted as `INTEGER`.

For scaling *D*, it is interpreted as one of the following depending on `mtype` (type of the matrix *A*) and *iparm(28)* (precision) specified in the preceding call to `cluster_sparse_solver`.

`DOUBLE PRECISION`	For real types of matrices (`mtype`=1, 2, -2, and 11) and for double precision (*iparm(28)* = 0).
`REAL`	For real types of matrices (`mtype`=1, 2, -2, and 11) and for single precision (*iparm(28)* = 1).
`DOUBLE COMPLEX`	For complex types of matrices (`mtype`=3, 6, 13, 14, and -4) and for double precision (*iparm(28)* = 0).
`COMPLEX`	For complex types of matrices (`mtype`=3, 6, 13, 14, and -4) and for single precision (*iparm(28)* = 1).

comm

`INTEGER*4`

MPI communicator. The solver uses the Fortran MPI communicator internally.

Output Parameters

error INTEGER

The error indicator:

error	Information
0	no error
-1	*pt* is a null pointer
-2	invalid *pt*
-3	invalid *name*
-4	unsupported *name*
-9	unsupported internal code path, consider switching off non-default *iparm* parameters for cluster_sparse_solver
-12	internal memory error

NOTE Refer to cl_solver_export_f90.f90 for an example using this functionality.

cluster_sparse_solver_export

Computes data corresponding to the specified decomposition (defined by export operation) and fills the pointers provided by calls to cluster_sparse_solver_set_ptr *and/or* cluster_sparse_solver_set_csr_ptrs.

Syntax

```
call cluster_sparse_solver_export (pt, operation, comm, error)
```

Include Files

- mkl_cluster_sparse_solver.f90

Description

This routine computes the data for the pointers of the (distributed) data to be exported (as defined the specified *operation*). It is assumed that the exported data will be distributed in the same way as the matrix *A* (as defined by *iparm(40)*) used in cluster_sparse_solver.

NOTE
Only call this routine after the factorization phase (*phase*=22) of the cluster_sparse_solver has been called. Neither *pt*, nor *iparm* should be changed after the preceding call to cluster_sparse_solver.

NOTE
Only call this routine after all pointers to the data required for the specified operation have been provided by means of calling `cluster_sparse_solver_set_ptr` and/or `cluster_sparse_solver_set_csr_ptrs`.

Input Parameters

pt

`INTEGER*8` for 64-bit architectures

Array with size of 64.

Handle to internal data structure used in the prior calls to `cluster_sparse_solver`.

Caution
Do not modify *pt* after the calls to `cluster_sparse_solver`.

operation

`INTEGER`

Specifies a particular operation which defines what data are exported

`SPARSE_PTLUQT`	Exporting data from decomposition $P*A*Q=L*U$.
`SPARSE_DPTLUQT`	Exporting data from decomposition from $P*(D^{-1}A)*Q=L*U$.

NOTE
Currently, for *operation*=`SPARSE_DPTLUQT` a real (complex) unit vector is provided for the scaling matrix *D*. Do not turn on scaling(*iparm(11)*>0) or matching(*iparm(13)*>0) in the *iparm* during the call to `cluster_sparse_solver` for this value of *operation*.

comm

`INTEGER*4`

MPI communicator. The solver uses the Fortran MPI communicator internally.

Output Parameters

error

`INTEGER`

The error indicator:

error	**Information**
`0`	no error
`-1`	*pt* is a null pointer
`-2`	invalid *pt*
`-5`	invalid *operation*

error	Information
-6	pointers to some of the data required for the specified *operation* were not provided prior to calling cluster_sparse_solver_export
-9	unsupported internal code path, consider switching off non-default *iparm* parameters for cluster_sparse_solver
-10	unsupported case when the matrix *A* is distributed among processes with overlap in the preceding calls to cluster_sparse_solver
-12	internal memory error

NOTE Refer to cl_solver_export_f90.f90 for an example using this functionality.

cluster_sparse_solver iparm Parameter

The following table describes all individual components of the Parallel Direct Sparse Solver for Clusters Interface *iparm* parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

Component	Description	
iparm(1) input	Use default values.	
	0	*iparm*(2) - *iparm*(64) are filled with default values.
	!=0	You must supply all values in components *iparm*(2) - *iparm*(64).
iparm(2) input	Fill-in reducing ordering for the input matrix.	
	2*	The nested dissection algorithm from the METIS package [Karypis98].
	3	The parallel version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Phase 1 takes significant time.
	10	The MPI version of the nested dissection and symbolic factorization algorithms for the matrix in distributed assembled matrix input format (iparm(40) > 0). The input matrix for the reordering must be distributed among different MPI processes without any intersection and all MPI ranks must have at least one row of the input matrix. Use iparm(41) and iparm(42) to set the bounds of the domain. During all of Phase 1, the entire matrix is not gathered on any one process, which can decrease computation time (especially when Phase 1 takes significant time) and decrease memory usage for each MPI process on the cluster. **NOTE** Distributed reordering does not work if any of matching(iparm(13)=1)/scaling(iparm(11)=1)/BSR format(iparm(37)>1)/Schur complement matrix computation control(iparm(36)>0)/Partial solve(iparm(31) > 0) is turned on, or if the distributed input matrix has overlapping distribution of rows across MPI processes.

Component	Description
	NOTE If you set *iparm*(2) = 10, *comm* = -1 (MPI communicator), and if there is one MPI process, optimization and full parallelization with the OpenMP version of the nested dissection and symbolic factorization algorithms proceeds. This can decrease computation time on multi-core computers. In this case, set *iparm*(41) = 1 and *iparm*(42) = *n* for one-based indexing, or to 0 and *n* – 1, respectively, for zero-based indexing.
iparm(3)	Reserved. Set to zero.
iparm(4)	Reserved. Set to zero.
iparm(5) input	User permutation. This parameter controls whether user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms. Another use of this parameter is to control obtaining the fill-in reducing permutation vector calculated during the reordering stage of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. This option is useful for testing reordering algorithms, adapting the code to special applications problems (for instance, to move zero diagonal elements to the end of $P*A*P^T$), or for using the permutation vector more than once for matrices with identical sparsity structures. For definition of the permutation, see the description of the *perm* parameter. **Caution** You can only set one of *iparm*(5), *iparm*(31), and *iparm*(36), so be sure that the *iparm*(31) (partial solution) and the *iparm*(36) (Schur complement) parameters are 0 if you set *iparm*(5).
	0 — User permutation in the *perm* array is ignored. 1 — Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the user supplied fill-in reducing permutation from the*perm* array. *iparm*(2) is ignored. 2 — Intel® oneAPI Math Kernel Library (oneMKL) PARDISO returns the permutation vector computed at phase 1 in the*perm* array.
iparm(6) input	Write solution on *x*. **NOTE** The array *x* is always used.
	0* — The array *x* contains the solution; right-hand side vector *b* is kept unchanged. 1 — The solver stores the solution on the right-hand side *b*.
iparm(7) output	Number of iterative refinement steps performed. Reports the number of iterative refinement steps that were actually performed during the solve step.
iparm(8) input	Iterative refinement step. On entry to the solve and iterative refinement step, *iparm*(8) must be set to the maximum number of iterative refinement steps that the solver performs.

Component	Description	
	0*	The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization.
	>0	Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of *iparm*(8) steps of iterative refinement. The solver might stop the process before the maximum number of steps if • a satisfactory level of accuracy of the solution in terms of backward error is achieved, • or if it determines that the required accuracy cannot be reached. In this case Parallel Direct Sparse Solver for Clusters Interface returns -4 in the *error* parameter. The number of executed iterations is reported in *iparm*(7).
	<0	Same as above, but the accumulation of the residuum uses extended precision real and complex data types. Perturbed pivots result in iterative refinement (independent of *iparm*(8)=0) and the number of executed iterations is reported in *iparm*(7).
iparm(9)	Reserved. Set to zero.	
iparm(10) input	Pivoting perturbation. This parameter instructs Parallel Direct Sparse Solver for Clusters Interface how to handle small pivots or zero pivots for nonsymmetric matrices (*mtype* =11 or *mtype* =13) and symmetric matrices (*mtype* =-2, *mtype* =-4, or *mtype* =6). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04]. Small pivots are perturbed with eps = $10^{-iparm(10)}$. The magnitude of the potential pivot is tested against a constant threshold of alpha = eps*$\|\|A2\|\|_{inf}$, where eps = $10^{(-iparm(10))}$, $A2 = P*P_{MPS}*D_r*A*D_c*P$, and $\|\|A2\|\|_{inf}$ is the infinity norm of the scaled and permuted matrix *A*. Any tiny pivots encountered during elimination are set to the sign (l_{II})*eps*$\|\|A2\|\|_{inf}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with eps = $10^{(-iparm(10))}$.	
	13*	The default value for nonsymmetric matrices(*mtype* =11, *mtype*=13), eps = 10^{-13}.
	8*	The default value for symmetric indefinite matrices (*mtype* =-2, *mtype*=-4, *mtype*=6), eps = 10^{-8}.
iparm(11) input	Scaling vectors. Parallel Direct Sparse Solver for Clusters Interface uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale. Use *iparm*(11) = 1 (scaling) and *iparm*(13) = 1 (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase (*phase*=11) you must provide the numerical values of the matrix *A* in array *a* in case of scaling and symmetric weighted matching.	
	0*	Disable scaling. Default for symmetric indefinite matrices.

Component	Description
	1* Enable scaling. Default for nonsymmetric matrices. Scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices (*mtype* = 11, *mtype* = 13). The scaling can also be used for symmetric indefinite matrices (*mtype* = -2, *mtype* = -4, *mtype* = 6) when the symmetric weighted matchings are applied (*iparm*(13) = 1). Note that in the analysis phase (*phase*=11) you must provide the numerical values of the matrix *A* in case of scaling.
iparm(12)	Solve with transposed or conjugate transposed matrix *A*. **NOTE** For real matrices, the terms *transposed* and *conjugate transposed* are equivalent. 0* Solve a linear system $AX = B$. 1 Solve a conjugate transposed system $A^HX = B$ based on the factorization of the matrix *A*. 2 Solve a transposed system $A^TX = B$ based on the factorization of the matrix *A*.
iparm(13) input	Improved accuracy using (non-) symmetric weighted matching. Parallel Direct Sparse Solver for Clusters Interface can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization. 0* Disable matching. Default for symmetric indefinite matrices. 1* Enable matching. Default for nonsymmetric matrices. Maximum weighted matching algorithm to permute large elements close to the diagonal. It is recommended to use *iparm*(11) = 1 (scaling) and *iparm*(13) = 1 (matching) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems. Note that in the analysis phase (*phase*=11) you must provide the numerical values of the matrix *A* in case of symmetric weighted matching.
iparm(14) output	Number of perturbed pivots. After factorization, contains the number of perturbed pivots for the matrix types: 1, 3, 11, 13, -2, -4 and 6.
iparm(15) output	Peak memory on symbolic factorization. The total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase. This value is only computed in phase 1.
iparm(16) output	Permanent memory on symbolic factorization. Permanent memory from the analysis and symbolic factorization phase in kilobytes that the solver needs in the factorization and solve phases.

Component	Description	
	This value is only computed in phase 1.	
iparm(17) output	Size of factors/Peak memory on numerical factorization and solution. This parameter provides the size in kilobytes of the total memory consumed by in-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for internal floating point arrays. This parameter is computed in phase 1. See *iparm*(63) for the OOC mode. The total peak memory consumed by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO is max(*iparm*(15), *iparm*(16)+*iparm*(17))	
iparm(18) input/output	Report the number of non-zero elements in the factors.	
	<0	Enable reporting if *iparm*(18) < 0 on entry. The default value is -1.
	>=0	Disable reporting.
iparm(19) - *iparm*(20)	Reserved. Set to zero.	
iparm(21) input	Pivoting for symmetric indefinite matrices.	
	0	Apply 1x1 diagonal pivoting during the factorization process.
	1*	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of *mtype*=-2, *mtype*=-4, or *mtype*=6.
iparm(22) output	Inertia: number of positive eigenvalues. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reports the number of positive eigenvalues for symmetric indefinite matrices.	
iparm(23) output	Inertia: number of negative eigenvalues. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reports the number of negative eigenvalues for symmetric indefinite matrices.	
iparm(24) - *iparm*(26)	Reserved. Set to zero.	
iparm(27) input	Matrix checker.	
	0*	Do not check the sparse matrix representation for errors.
	1	Check integer arrays *ia* and *ja*. In particular, check whether the column indices are sorted in increasing order within each row.
iparm(28) input	Single or double precision Parallel Direct Sparse Solver for Clusters Interface. See *iparm*(8) for information on controlling the precision of the refinement steps.	
	0*	Input arrays (*a*, *x* and *b*) and all internal arrays must be presented in double precision.
	1	Input arrays (*a*, *x* and *b*) must be presented in single precision. In this case all internal computations are performed in single precision.
iparm(29)	Reserved. Set to zero.	
iparm(30) output	Number of zero or negative pivots.	

Component	Description
	If Intel® oneAPI Math Kernel Library (oneMKL) PARDISO detects zero or negative pivot for*mtype*=2 or *mtype*=4 matrix types, the factorization is stopped. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO returns immediately with an*error* = -4, and *iparm*(30) reports the number of the equation where the zero or negative pivot is detected. Note: The returned value can be different for the parallel and sequential version in case of several zero/negative pivots.
iparm(31) input	Partial solve and computing selected components of the solution vectors. This parameter controls the solve step of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. It can be used if only a few components of the solution vectors are needed or if you want to reduce the computation cost at the solve step by utilizing the sparsity of the right-hand sides. To use this option the input permutation vector define*perm* so that when *perm*(*i*) = 1 it means that either the *i*-th component in the right-hand sides is nonzero, or the *i*-th component in the solution vectors is computed, or both, depending on the value of *iparm*(31). The permutation vector *perm*must be present in all phases of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO software. At the reordering step, the software overwrites the input vector*perm* by a permutation vector used by the software at the factorization and solver step. If *m* is the number of components such that *perm*(*i*) = 1, then the last *m* components of the output vector *perm* are a set of the indices *i* satisfying the condition *perm*(*i*) = 1 on input. **NOTE** Turning on this option often increases the time used by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for factorization and reordering steps, but it can reduce the time required for the solver step. **Important** Set the parameters *iparm*(8) (iterative refinement steps), *iparm*(4) (preconditioned CGS), *iparm*(5) (user permutation), and *iparm*(36) (Schur complement) to 0 as well.

0*	Disables this option.
1	it is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector *perm* is defined so that *perm*(*i*) = 1 means that the (*i*)-th component in the right-hand sides is nonzero. In this case Intel® oneAPI Math Kernel Library (oneMKL) PARDISO only uses the non-zero components of the right-hand side vectors and computes only corresponding components in the solution vectors. That means the*i*-th component in the solution vectors is only computed if *perm*(*i*) = 1.
2	It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector *perm* is defined so that *perm*(*i*) = 1 means that the *i*-th component in the right-hand sides is nonzero. Unlike for *iparm*(31)=1, all components of the solution vector are computed for this setting and all components of the right-hand sides are used. Because all components are used, for *iparm*(31)=2 you must set the *i*-th component of the right-hand sides to zero explicitly if *perm*(*i*) is not equal to 1.

Component	Description	
	3	Selected components of the solution vectors are computed. The *perm* array is not related to the right-hand sides and it only indicates which components of the solution vectors should be computed. In this case *perm*(*i*) = 1 means that the *i*-th component in the solution vectors is computed.
iparm(31) - *iparm*(33)	Reserved. Set to zero.	
iparm(35) input	One- or zero-based indexing of columns and rows.	
	0*	One-based indexing: columns and rows indexing in arrays *ia*, *ja*, and *perm* starts from 1 (Fortran-style indexing).
	1	Zero-based indexing: columns and rows indexing in arrays *ia*, *ja*, and *perm* starts from 0 (C-style indexing).
iparm(36) input	Schur complement matrix computation control. To calculate this matrix, you must set the input permutation vector *perm* to a set of indexes such that when *perm*(*i*) = 1, the *i*-th element of the initial matrix is an element of the Schur matrix. **Caution** You can only set one of *iparm*(5), *iparm*(31), and *iparm*(36), so be sure that the *iparm*(5) (user permutation) and the *iparm*(31) (partial solution) parameters are 0 if you set *iparm*(36).	
	0*	Do not compute Schur complement.
	1	Compute Schur complement matrix as part of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factorization step and return it in the solution vector. **NOTE** This option only computes the Schur complement matrix, and does not calculate factorization arrays.
	2	Compute Schur complement matrix as part of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factorization step and return it in the solution vector. Since this option calculates factorization arrays you can use it to launch partial or full solution of the entire problem after the factorization step.
iparm(37) input	Format for matrix storage.	
	0*	Use CSR format (see Three Array Variation of BSR Format) for matrix storage.
	1	Use CSR format (see Three Array Variation of BSR Format) for matrix storage.
	< 0	Convert supplied matrix to variable BSR (VBSR) format (see Sparse Data Storage) for matrix storage. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO analyzes the matrix provided in CSR3 format and converts it to an internal VBSR format. Set*iparm*(37) = $-t$, $0 < t \leq 100$.
iparm(38) - *iparm*(39)	Reserved. Set to zero.	
iparm(40) input	Matrix input format.	

Component	Description
	NOTE Performance of the reordering step of the Parallel Direct Sparse Solver for Clusters Interface is slightly better for assembled format (CSR, *iparm*(40) = 0) than for distributed format (DCSR, *iparm*(40) > 0) for the same matrices, so if the matrix is assembled on one node do not distribute it before calling cluster_sparse_solver.
	0* — Provide the matrix in usual centralized input format: the master MPI process stores all data from matrix *A*, with rank=0.
	1 — Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix *A* data. Set the bounds of the domain using *iparm*(41) and *iparm*(42). The solution vector is placed on the master process.
	2 — Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix *A* data. Set the bounds of the domain using *iparm*(41) and *iparm*(42). The solution vector, *A*, and RHS elements are distributed between processes in same manner.
	3 — Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix *A* data. Set the bounds of the domain using *iparm*(41) and *iparm*(42). The *A* and RHS elements are distributed between processes in same manner and the solution vector is the same on each process
iparm(41) input	Beginning of input domain. The number of the matrix *A* row, RHS element, and, for *iparm*(40)=2, solution vector that begins the input domain belonging to this MPI process. Only applicable to the distributed assembled matrix input format (*iparm*(40)> 0). See Sparse Matrix Storage Formats for more details.
iparm(42) input	End of input domain. The number of the matrix *A* row, RHS element, and, for *iparm*(40)=2, solution vector that ends the input domain belonging to this MPI process. Only applicable to the distributed assembled matrix input format (*iparm*(40)> 0). See Sparse Matrix Storage Formats for more details.
iparm(43) - *iparm*(59) input	Reserved. Set to zero.
iparm(60) input	cluster_sparse_solver mode. *iparm*(60) switches between in-core (IC) and out-of-core (OOC) of cluster_sparse_solver. OOC can solve very large problems by holding the matrix factors in files on the disk, which requires a reduced amount of main memory compared to IC. Unless you are operating in sequential mode, you can switch between IC and OOC modes after the reordering phase. However, you can get better cluster_sparse_solver performance by setting *iparm*(60) before the reordering phase. The amount of memory used in OOC mode depends on the number of OpenMP threads.

Component	Description
	Warning Do not increase the number of OpenMP threads used for cluster_sparse_solver between the first call and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.
	0* IC mode.
	1 IC mode is used if the total amount of RAM (in megabytes) needed for storing the matrix factors is less than sum of two values of the environment variables: `MKL_PARDISO_OOC_MAX_CORE_SIZE` (default value 2000 MB) and `MKL_PARDISO_OOC_MAX_SWAP_SIZE` (default value 0 MB); otherwise OOC mode is used. In this case amount of RAM used by OOC mode cannot exceed the value of `MKL_PARDISO_OOC_MAX_CORE_SIZE`. If the total peak memory needed for storing the local arrays is more than `MKL_PARDISO_OOC_MAX_CORE_SIZE`, increase `MKL_PARDISO_OOC_MAX_CORE_SIZE` if possible. **NOTE** Conditional numerical reproducibility (CNR) is not supported for this mode.
	2 OOC mode. The OOC mode can solve very large problems by holding the matrix factors in files on the disk. Hence the amount of RAM required by OOC mode is significantly reduced compared to IC mode. If the total peak memory needed for storing the local arrays is more than `MKL_PARDISO_OOC_MAX_CORE_SIZE`, increase `MKL_PARDISO_OOC_MAX_CORE_SIZE` if possible. To obtain better cluster_sparse_solver performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase. **NOTE** To use OOC mode, you must disable *iparm*(11) (scaling) and *iparm*(13) = 1 (matching).
iparm(61) - *iparm*(62) input	Reserved. Set to zero.
iparm(63) output	Size of the minimum OOC memory for numerical factorization and solution. This parameter provides the size in kilobytes of the minimum memory required by OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for internal floating point arrays. This parameter is computed in phase 1. Total peak memory consumption of OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO can be estimated as max(*iparm*(15), *iparm*(16) + *iparm*(63)).
iparm(64) input	Reserved. Set to zero.

NOTE
Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See Sparse Matrix Storage Formats for an example.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Direct Sparse Solver (DSS) Interface Routines

Intel® oneAPI Math Kernel Library (oneMKL) supports the DSS interface, an alternative to the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO interface for the direct sparse solver. The DSS interface implements a group of user-callable routines that are used in the step-by-step solving process and utilizes the general scheme described inAppendix A Linear Solvers Basics for solving sparse systems of linear equations. This interface also includes one routine for gathering statistics related to the solving process and an auxiliary routine for passing character strings from Fortran routines to C routines.

The DSS interface also supports the out-of-core (OOC) mode.

Table "DSS Interface Routines" lists the names of the routines and describes their general use.

DSS Interface Routines

Routine	Description
dss_create	Initializes the solver and creates the basic data structures necessary for the solver. This routine must be called before any other DSS routine.
dss_define_structure	Informs the solver of the locations of the non-zero elements of the matrix.
dss_reorder	Based on the non-zero structure of the matrix, computes a permutation vector to reduce fill-in during the factoring process.
dss_factor_real, dss_factor_complex	Computes the LU, LDL^T or LL^T factorization of a real or complex matrix.
dss_solve_real, dss_solve_complex	Computes the solution vector for a system of equations based on the factorization computed in the previous phase.
dss_delete	Deletes all data structures created during the solving process.
dss_statistics	Returns statistics about various phases of the solving process.
mkl_cvt_to_null_terminated_str	Passes character strings from Fortran routines to C routines.

To find a single solution vector for a single system of equations with a single right-hand side, invoke the Intel® oneAPI Math Kernel Library (oneMKL) DSS interface routines in this order:

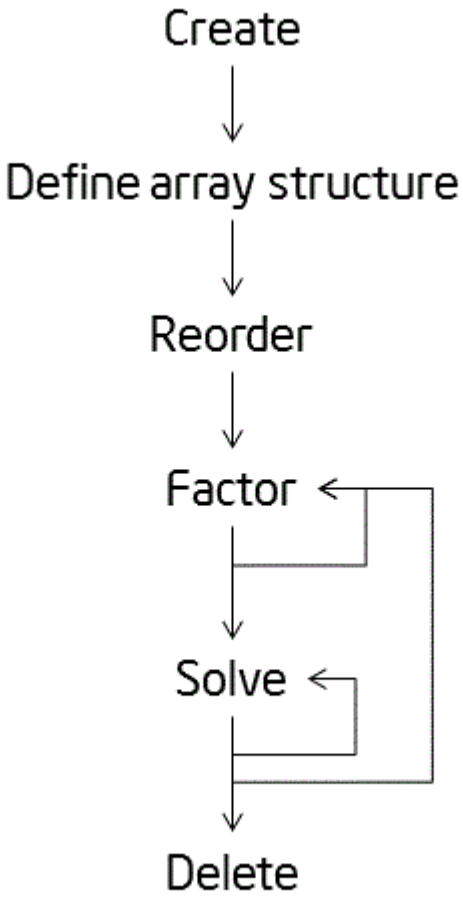
1. dss_create
2. dss_define_structure
3. dss_reorder
4. dss_factor_real, dss_factor_complex

5. dss_solve_real, dss_solve_complex
6. dss_delete

However, in certain applications it is necessary to produce solution vectors for multiple right-hand sides for a given factorization and/or factor several matrices with the same non-zero structure. Consequently, it is sometimes necessary to invoke the Intel® oneAPI Math Kernel Library (oneMKL) sparse routines in an order other than that listed, which is possible using the DSS interface. The solving process is conceptually divided into six phases.Figure "Typical order for invoking DSS interface routines" indicates the typical order in which the DSS interface routines can be invoked.

__border__top

Typical order for invoking DSS interface routines



See the code examples that use the DSS interface routines to solve systems of linear equations in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory (`dss_*.f`).

- `examples/solverf/source`

DSS Interface Description

Each DSS routine reads from or writes to a data object called a *handle*. Refer to Memory Allocation and Handles to determine the correct method for declaring a handle argument for each language. For simplicity, the descriptions in DSS routines refer to the data type as `MKL_DSS_HANDLE`.

Routine Options

The DSS routines have an integer argument (referred below to as *opt*) for passing various options to the routines. The permissible values for *opt* should be specified using only the symbol constants defined in the language-specific header files (see Implementation Details). The routines accept options for setting the message and termination levels as described in Table "Symbolic Names for the Message and Termination Levels Options". Additionally, each routine accepts the option `MKL_DSS_DEFAULTS` that sets the default values (as documented) for *opt* to the routine.

Symbolic Names for the Message and Termination Levels Options

Message Level	Termination Level
MKL_DSS_MSG_LVL_SUCCESS	MKL_DSS_TERM_LVL_SUCCESS
MKL_DSS_MSG_LVL_INFO	MKL_DSS_TERM_LVL_INFO
MKL_DSS_MSG_LVL_WARNING	MKL_DSS_TERM_LVL_WARNING
MKL_DSS_MSG_LVL_ERROR	MKL_DSS_TERM_LVL_ERROR
MKL_DSS_MSG_LVL_FATAL	MKL_DSS_TERM_LVL_FATAL

The settings for message and termination levels can be set on any call to a DSS routine. However, once set to a particular level, they remain at that level until they are changed in another call to a DSS routine.

You can specify both message and termination level for a DSS routine by adding the options together. For example, to set the message level to `debug` and the termination level to `error` for all the DSS routines, use the following call:

```
CALL dss_create( handle, MKL_DSS_MSG_LVL_INFO + MKL_DSS_TERM_LVL_ERROR)
```

User Data Arrays

Many of the DSS routines take arrays of user data as input. For example, user arrays are passed to the routine `dss_define_structure` to describe the location of the non-zero entries in the matrix.

Caution
Do not modify the contents of these arrays after they are passed to one of the solver routines.

DSS Implementation Details

To promote portability across platforms and ease of use across different languages, use the `mkl_dss.f90` header file.

The header file defines symbolic constants for returned error values, function options, certain defined data types, and function prototypes.

NOTE
Constants for options, returned error values, and message severities must be referred only by the symbolic names that are defined in these header files. Use of the Intel® oneAPI Math Kernel Library (oneMKL) DSS software without including one of the above header files is not supported.

Memory Allocation and Handles

You do not need to allocate any temporary working storage in order to use the Intel® oneAPI Math Kernel Library (oneMKL) DSS routines, because the solver itself allocates any required storage. To enable multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using a*handle* data object.

Each of the Intel® oneAPI Math Kernel Library (oneMKL) DSS routines creates, uses, or deletes a handle. Consequently, any program calling an Intel® oneAPI Math Kernel Library (oneMKL) DSS routine must be able to allocate storage for a handle. The exact syntax for allocating storage for a handle varies from language to language. To standardize the handle declarations, the language-specific header files declare constants and defined data types that must be used when declaring a handle object in your code.

```
INCLUDE "mkl_dss.f90"
TYPE(MKL_DSS_HANDLE) handle
```

In addition to the definition for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the returned error values
- user options for the solver routines
- constants indicating message severity.

DSS Routines

dss_create

Initializes the solver.

Syntax

`call dss_create(handle, opt)`

Include Files

- `mkl.fi`, `mkl_dss.f90`

Description

The `dss_create` routine initializes the solver. After the call to `dss_create`, all subsequent invocations of the Intel® oneAPI Math Kernel Library (oneMKL) DSS routines must use the value of the handle returned by`dss_create`.

> **WARNING**
> Do not write the value of handle directly.

The default value of the parameter *opt* is

`MKL_DSS_MSG_LVL_WARNING` + `MKL_DSS_TERM_LVL_ERROR`.

By default, the DSS routines use double precision for solving systems of linear equations. The precision used by the DSS routines can be set to single mode by adding the following value to the *opt* parameter:

`MKL_DSS_SINGLE_PRECISION`.

Input data and internal arrays are required to have single precision.

By default, the DSS routines use Fortran style (one-based) indexing for input arrays of integer types (the first value is referenced as array element 1). To set indexing to C style (the first value is referenced as array element 0), add the following value to the *opt* parameter:

MKL_DSS_ZERO_BASED_INDEXING.

The *opt* parameter can also control number of refinement steps used on the solution stage by specifying the two following values:

MKL_DSS_REFINEMENT_OFF - maximum number of refinement steps is set to zero;
MKL_DSS_REFINEMENT_ON (default value) - maximum number of refinement steps is set to 2.

By default, DSS uses in-core computations. To launch the out-of-core version of DSS (OOC DSS) you can add to this parameter one of two possible values: MKL_DSS_OOC_STRONG and MKL_DSS_OOC_VARIABLE.

MKL_DSS_OOC_STRONG - OOC DSS is used.

MKL_DSS_OOC_VARIABLE - if the memory needed for the matrix factors is less than the value of the environment variable MKL_PARDISO_OOC_MAX_CORE_SIZE, then the OOC DSS uses the in-core kernels of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO, otherwise it uses the OOC computations.

The variable MKL_PARDISO_OOC_MAX_CORE_SIZE defines the maximum size of RAM allowed for storing work arrays associated with the matrix factors. It is ignored if MKL_DSS_OOC_STRONG is set. The default value of MKL_PARDISO_OOC_MAX_CORE_SIZE is 2000 MB. This value and default path and file name for storing temporary data can be changed using the configuration file pardiso_ooc.cfg or command line (See more details in the description of the pardiso routine).

WARNING
Other than message and termination level options, do not change the OOC DSS settings after they are specified in the routine dss_create.

Input Parameters

opt

INTEGER, INTENT(IN)

Parameter to pass the DSS options. The default value is MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR.

Output Parameters

handle

TYPE (MKL_DSS_HANDLE), INTENT(OUT)

Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

Return Values

MKL_DSS_SUCCESS

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_MSG_LVL_ERR

MKL_DSS_TERM_LVL_ERR

dss_define_structure

Communicates locations of non-zero elements in the matrix to the solver.

Syntax

```
call dss_define_structure(handle, opt, rowIndex, nRows, nCols, columns, nNonZeros);
```

Include Files

- mkl.fi, mkl_dss.f90

Description

The routine dss_define_structure communicates the locations of the *nNonZeros* number of non-zero elements in a matrix of *nRows * nCols* size to the solver.

> **NOTE**
> The Intel® oneAPI Math Kernel Library (oneMKL) DSS software operates only on square matrices, so*nRows* must be equal to *nCols*.

To communicate the locations of non-zero elements in the matrix, do the following:

1. Define the general non-zero structure of the matrix by specifying the value for the options argument *opt*. You can set the following values for real matrices:
   - MKL_DSS_SYMMETRIC_STRUCTURE
   - MKL_DSS_SYMMETRIC
   - MKL_DSS_NON_SYMMETRIC

   and for complex matrices:
   - MKL_DSS_SYMMETRIC_STRUCTURE_COMPLEX
   - MKL_DSS_SYMMETRIC_COMPLEX
   - MKL_DSS_NON_SYMMETRIC_COMPLEX

   The information about the matrix type must be defined in dss_define_structure.
2. Provide the actual locations of the non-zeros by means of the arrays *rowIndex* and *columns* (see Sparse Matrix Storage Format).

> **NOTE** No diagonal element can be omitted from the values array. If there is a zero value on the diagonal, for example, that element nonetheless must be explicitly represented.

Input Parameters

opt — INTEGER, INTENT(IN)

Parameter to pass the DSS options. The default value for the matrix structure is MKL_DSS_SYMMETRIC.

rowIndex — INTEGER, INTENT(IN)

Array of size *nRows*+1. Defines the location of non-zero entries in the matrix.

nRows — INTEGER, INTENT(IN)

Number of rows in the matrix.

nCols — INTEGER, INTENT(IN)

Number of columns in the matrix; must be equal to *nRows*.

columns	INTEGER, INTENT(IN) Array of size *nNonZeros*. Defines the column location of non-zero entries in the matrix.
nNonZeros	INTEGER, INTENT(IN) Number of non-zero elements in the matrix.

Output Parameters

handle	TYPE (MKL_DSS_HANDLE), INTENT(INOUT) Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_STRUCTURE_ERR

MKL_DSS_ROW_ERR

MKL_DSS_COL_ERR

MKL_DSS_NOT_SQUARE

MKL_DSS_TOO_FEW_VALUES

MKL_DSS_TOO_MANY_VALUES

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_MSG_LVL_ERR

MKL_DSS_TERM_LVL_ERR

dss_reorder

Computes or sets a permutation vector that minimizes the fill-in during the factorization phase.

Syntax

```
call dss_reorder(handle, opt, perm)
```

Include Files

- mkl.fi, mkl_dss.f90

Description

If *opt* contains the option MKL_DSS_AUTO_ORDER, then the routine dss_reorder computes a permutation vector that minimizes the fill-in during the factorization phase. For this option, the routine ignores the contents of the *perm* array.

If *opt* contains the option MKL_DSS_METIS_OPENMP_ORDER, then the routine dss_reorder computes permutation vector using the parallel nested dissections algorithm to minimize the fill-in during the factorization phase. This option can be used to decrease the time of dss_reorder call on multi-core computers. For this option, the routine ignores the contents of the *perm* array.

If *opt* contains the option MKL_DSS_MY_ORDER, then you must supply a permutation vector in the array *perm*. In this case, the array *perm* is of length *nRows*, where *nRows* is the number of rows in the matrix as defined by the previous call to dss_define_structure.

If *opt* contains the option MKL_DSS_GET_ORDER, then the permutation vector computed during the dss_reorder call is copied to the array *perm*. In this case you must allocate the array *perm* beforehand. The permutation vector is computed in the same way as if the option MKL_DSS_AUTO_ORDER is set.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

opt

INTEGER, INTENT(IN)

Parameter to pass the DSS options. The default value for the permutation type is MKL_DSS_AUTO_ORDER.

perm

INTEGER, INTENT(IN)

Array of length *nRows*. Contains a user-defined permutation vector (accessed only if *opt* contains MKL_DSS_MY_ORDER or MKL_DSS_GET_ORDER).

Output Parameters

handle

TYPE (MKL_DSS_HANDLE), INTENT(INOUT)

Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_REORDER_ERR

MKL_DSS_REORDER1_ERR

MKL_DSS_I32BIT_ERR

MKL_DSS_FAILURE

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_MSG_LVL_ERR

MKL_DSS_TERM_LVL_ERR

dss_factor_real, dss_factor_complex

Compute factorization of the matrix with previously specified location of non-zero elements.

Syntax

```
call dss_factor_real(handle, opt, rValues)
call dss_factor_complex(handle, opt, cValues)
call dss_factor(handle, opt, Values)
```

Include Files

- mkl.fi, mkl_dss.f90

Description

These routines compute factorization of the matrix whose non-zero locations were previously specified by a call to dss_define_structure and whose non-zero values are given in the array *rValues*, *cValues* or *Values*. Data type These arrays must be of length *nNonZeros* as defined in a previous call to dss_define_structure.

> **NOTE**
> The data type (single or double precision) of *rValues*, *cValues*, *Values* must be in correspondence with precision specified by the parameter *opt* in the routine dss_create.

The *opt* argument can contain one of the following options:

- MKL_DSS_POSITIVE_DEFINITE
- MKL_DSS_INDEFINITE
- MKL_DSS_HERMITIAN_POSITIVE_DEFINITE
- MKL_DSS_HERMITIAN_INDEFINITE

depending on your matrix's type.

> **NOTE**
> This routine supports the Progress Routine feature. See Progress Function for details.

Input Parameters

handle

TYPE (MKL_DSS_HANDLE), INTENT(INOUT)

Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

opt

INTEGER, INTENT(IN)

Parameter to pass the DSS options. The default value is MKL_DSS_POSITIVE_DEFINITE.

rValues

REAL*8

REAL(KIND=4), INTENT(IN) or

REAL(KIND=8), INTENT(IN)

Array of elements of the matrix *A*. Real data, single or double precision as it is specified by the parameter *opt* in the routine dss_create.

cValues

COMPLEX*16

	`COMPLEX(KIND=4), INTENT(IN)` or `COMPLEX(KIND=8), INTENT(IN)` Array of elements of the matrix *A*. Complex data, single or double precision as it is specified by the parameter *opt* in the routine `dss_create`.
Values	`REAL(KIND=4), INTENT(OUT),` or `REAL(KIND=8), INTENT(OUT),` or `COMPLEX(KIND=4), INTENT(OUT),` or `COMPLEX(KIND=8), INTENT(OUT)` Array of elements of the matrix *A*. Real or complex data, single or double precision as it is specified by the parameter *opt* in the routine `dss_create`.

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_STATE_ERR`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_OPTION_CONFLICT`

`MKL_DSS_VALUES_ERR`

`MKL_DSS_OUT_OF_MEMORY`

`MKL_DSS_ZERO_PIVOT`

`MKL_DSS_FAILURE`

`MKL_DSS_MSG_LVL_ERR`

`MKL_DSS_TERM_LVL_ERR`

`MKL_DSS_OOC_MEM_ERR`

`MKL_DSS_OOC_OC_ERR`

`MKL_DSS_OOC_RW_ERR`

dss_solve_real, dss_solve_complex

Compute the corresponding solution vector and place it in the output array.

Syntax

```
call dss_solve_real(handle, opt, rRhsValues, nRhs, rSolValues)
call dss_solve_complex(handle, opt, cRhsValues, nRhs, cSolValues)
call dss_solve(handle, opt, RhsValues, nRhs, SolValues)
```

Include Files

- `mkl.fi, mkl_dss.f90`

Description

For each right-hand side column vector defined in the arrays *rRhsValues*, *cRhsValues*, or *RhsValues*, these routines compute the corresponding solution vector and place it in the arrays *rSolValues*, *cSolValues*, or *SolValues* respectively.

> **NOTE**
> The data type (single or double precision) of all arrays must be in correspondence with precision specified by the parameter *opt* in the routine dss_create.

The lengths of the right-hand side and solution vectors, *nRows* and *nCols* respectively, must be defined in a previous call to dss_define_structure.

By default, both routines perform the full solution step (it corresponds to *phase* = 33in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO). The parameter*opt* enables you to calculate the final solution step-by-step, calling forward and backward substitutions.

If it is set to MKL_DSS_FORWARD_SOLVE, the forward substitution (corresponding to *phase* = 331in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO) is performed;

if it is set to MKL_DSS_DIAGONAL_SOLVE, the diagonal substitution (corresponding to *phase* = 332in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO) is performed, if possible;

if it is set to MKL_DSS_BACKWARD_SOLVE, the backward substitution (corresponding to *phase* = 333in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO) is performed.

For more details about using these substitutions for different types of matrices, see Separate Forward and Backward Substitutionin the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver description.

This parameter also can control the number of refinement steps that is used on the solution stage: if it is set to MKL_DSS_REFINEMENT_OFF, the maximum number of refinement steps equal to zero, and if it is set to MKL_DSS_REFINEMENT_ON (default value), the maximum number of refinement steps is equal to 2.

MKL_DSS_CONJUGATE_SOLVE option added to the parameter *opt* enables solving a conjugate transposed system $A^HX = B$ based on the factorization of the matrix A. This option is equivalent to the parameter *iparm*(12)= 1in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

MKL_DSS_TRANSPOSE_SOLVE option added to the parameter *opt* enables solving a transposed system $A^TX = B$ based on the factorization of the matrix A. This option is equivalent to the parameter *iparm*(12)= 2in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Input Parameters

handle

TYPE (MKL_DSS_HANDLE), INTENT(INOUT)

Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

opt

INTEGER, INTENT(IN)

Parameter to pass the DSS options.

nRhs

INTEGER, INTENT(IN)

Number of the right-hand sides in the system of linear equations.

rRhsValues

REAL*8

REAL(KIND=4), INTENT(IN) or

REAL(KIND=8), INTENT(IN)

Array of size *nRows* * *nRhs*. Contains real right-hand side vectors. Real data, single or double precision as it is specified by the parameter *opt* in the routine dss_create.

cRhsValues

COMPLEX*16

COMPLEX(KIND=4), INTENT(IN) or

COMPLEX(KIND=8), INTENT(IN)

Array of size *nRows* * *nRhs*. Contains complex right-hand side vectors. Complex data, single or double precision as it is specified by the parameter *opt* in the routine dss_create.

RhsValues

REAL(KIND=4), INTENT(IN), or

REAL(KIND=8), INTENT(IN), or

COMPLEX(KIND=4), INTENT(IN), or

COMPLEX(KIND=8), INTENT(IN)

Array of size *nRows* * *nRhs*. Contains right-hand side vectors. Real or complex data, single or double precision as it is specified by the parameter *opt* in the routine dss_create.

Output Parameters

rSolValues

REAL(KIND=4), INTENT(OUT) or

REAL(KIND=8), INTENT(OUT)

Array of size *nCols* * *nRhs*. Contains real solution vectors. Real data, single or double precision as it is specified by the parameter *opt* in the routine dss_create.

cSolValues

COMPLEX(KIND=4), INTENT(OUT) or

COMPLEX(KIND=8), INTENT(OUT)

Array of size *nCols* * *nRhs*. Contains complex solution vectors. Complex data, single or double precision as it is specified by the parameter *opt* in the routine dss_create.

SolValues

REAL(KIND=4), INTENT(OUT), or

REAL(KIND=8), INTENT(OUT), or

COMPLEX(KIND=4), INTENT(OUT), or

COMPLEX(KIND=8), INTENT(OUT)

Array of size *nCols* * *nRhs*. Contains solution vectors. Real or complex data, single or double precision as it is specified by the parameter *opt* in the routine dss_create.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_DIAG_ERR

MKL_DSS_FAILURE

MKL_DSS_MSG_LVL_ERR

MKL_DSS_TERM_LVL_ERR

MKL_DSS_OOC_MEM_ERR

MKL_DSS_OOC_OC_ERR

MKL_DSS_OOC_RW_ERR

dss_delete

Deletes all of the data structures created during the solutions process.

Syntax

`call dss_delete(handle, opt)`

Include Files

- `mkl.fi, mkl_dss.f90`

Description

The routine `dss_delete` deletes all data structures created during the solving process.

Input Parameters

opt

`INTEGER, INTENT(IN)`

Parameter to pass the DSS options. The default value is `MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

Output Parameters

handle

`TYPE (MKL_DSS_HANDLE), INTENT(INOUT)`

Pointer to the data structure storing internal DSS results (`MKL_DSS_HANDLE`).

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_MSG_LVL_ERR

MKL_DSS_TERM_LVL_ERR

dss_statistics

Returns statistics about various phases of the solving process.

Syntax

```
call dss_statistics(handle, opt, statArr, retValues)
```

Include Files

- mkl.fi, mkl_dss.f90

Description

The dss_statistics routine returns statistics about various phases of the solving process. This routine gathers the following statistics:

- time taken to do reordering,
- time taken to do factorization,
- duration of problem solving,
- determinant of the symmetric indefinite input matrix,
- inertia of the symmetric indefinite input matrix,
- number of floating point operations taken during factorization,
- total peak memory needed during the analysis and symbolic factorization,
- permanent memory needed from the analysis and symbolic factorization,
- memory consumption for the factorization and solve phases.

Statistics are returned in accordance with the input string specified by the parameter *statArr*. The value of the statistics is returned in double precision in a return array, which you must allocate beforehand.

For multiple statistics, multiple string constants separated by commas can be used as input. Return values are put into the return array in the same order as specified in the input string.

Statistics can only be requested at the appropriate stages of the solving process. For example, requesting FactorTime before a matrix is factored leads to an error.

The following table shows the point at which each individual statistics item can be requested:

Statistics Calling Sequences

Type of Statistics	When to Call
ReorderTime	After dss_reorder is completed successfully.
FactorTime	After dss_factor_real or dss_factor_complex is completed successfully.
SolveTime	After dss_solve_real or dss_solve_complex is completed successfully.
Determinant	After dss_factor_real or dss_factor_complex is completed successfully.
Inertia	After dss_factor_real is completed successfully and the matrix is real, symmetric, and indefinite.
Flops	After dss_factor_real or dss_factor_complex is completed successfully.
Peakmem	After dss_reorder is completed successfully.
Factormem	After dss_reorder is completed successfully.
Solvemem	After dss_factor_real or dss_factor_complex is completed successfully.

Input Parameters

handle — TYPE (MKL_DSS_HANDLE), INTENT(IN)

Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

opt

INTEGER, INTENT(IN)

Parameter to pass the DSS options.

statArr

INTEGER, INTENT(IN)

Input string that defines the type of the returned statistics. The parameter can include one or more of the following string constants (case of the input string has no effect):

`ReorderTime`	Amount of time taken to do the reordering.
`FactorTime`	Amount of time taken to do the factorization.
`SolveTime`	Amount of time taken to solve the problem after factorization.
`Determinant`	Determinant of the matrix *A*. For real matrices: the determinant is returned as `det_pow`, `det_base` in two consecutive return array locations, where `1.0` ≤ `abs(`*det_base*`) < 10.0` and `determinant =` *det_base*`*10`$^{(det\_pow)}$. For complex matrices: the determinant is returned as *det_pow*, *det_re*, *det_im* in three consecutive return array locations, where `1.0` ≤`abs(`*det_re*`) + abs(`*det_im*`) < 10.0` and `determinant = (`*det_re*`,` *det_im*`)*10`$^{(det\_pow)}$.
`Inertia`	Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers `(`*p*`,`*n*`,`*z*`)`, where *p* is the number of positive eigenvalues, *n* is the number of negative eigenvalues, and *z* is the number of zero eigenvalues. `Inertia` is returned as three consecutive return array locations *p*, *n*, *z*. Computing inertia can lead to incorrect results for matrixes with a cluster of eigenvalues which are near 0. `Inertia` of a *k*-by-*k* real symmetric positive definite matrix is always `(`*k*`, 0, 0)`. Therefore `Inertia` is returned only in cases of real symmetric indefinite matrices. For all other matrix types, an error message is returned.
`Flops`	Number of floating point operations performed during the factorization.
`Peakmem`	Total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase.

`Factormem`	Permanent memory in kilobytes that the solver needs from the analysis and symbolic factorization phase in the factorization and solve phases.
`Solvemem`	Total double precision memory consumption (kilobytes) of the solver for the factorization and solve phases.

NOTE
To avoid problems in passing strings from Fortran to C, Fortran users must call the `mkl_cvt_to_null_terminated_str` routine before calling `dss_statistics`. Refer to the description of `mkl_cvt_to_null_terminated_str` for details.

Output Parameters

`retValues`	`REAL(KIND=8), INTENT(OUT)` Value of the statistics returned.

Finding 'time used to reorder' and 'inertia' of a matrix

The example below illustrates the use of the `dss_statistics` routine.

To find the above values, call `dss_statistics`(*`handle`*, *`opt`*, *`statArr`*, *`retValue`*), where *`staArr`* is "`ReorderTime,Inertia`"

In this example, *`retValue`* has the following values:

`retValue``[0]`	Time to reorder.
`retValue``[1]`	Positive Eigenvalues.
`retValue``[2]`	Negative Eigenvalues.
`retValue``[3]`	Zero Eigenvalues.

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_STATISTICS_INVALID_MATRIX`

`MKL_DSS_STATISTICS_INVALID_STATE`

`MKL_DSS_STATISTICS_INVALID_STRING`

`MKL_DSS_MSG_LVL_ERR`

`MKL_DSS_TERM_LVL_ERR`

mkl_cvt_to_null_terminated_str

Passes character strings from Fortran routines to C routines.

Syntax

`mkl_cvt_to_null_terminated_str (`*`destStr`*`, `*`destLen`*`, `*`srcStr`*`)`

Include Files

- `mkl.fi`, `mkl_dss.f90`

Description

The routine `mkl_cvt_to_null_terminated_str` passes character strings from Fortran routines to C routines. The strings are converted into integer arrays before being passed to C. Using this routine avoids the problems that can occur on some platforms when passing strings from Fortran to C. The use of this routine is highly recommended.

Input Parameters

destLen	`INTEGER`. Length of the output array *destStr*.
srcStr	`STRING`. Input string.

Output Parameters

destStr	`INTEGER`. One-dimensional array of integers.

Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)

Intel® oneAPI Math Kernel Library (oneMKL) supports iterative sparse solvers (ISS) based on the reverse communication interface (RCI), referred to here as the RCI ISS interface. The RCI ISS interface implements a group of user-callable routines that are used in the step-by-step solving process of a symmetric positive definite system (RCI conjugate gradient solver, or RCI CG), and of a non-symmetric indefinite (non-degenerate) system (RCI flexible generalized minimal residual solver, or RCI FGMRES) of linear algebraic equations. This interface uses the general RCI scheme described in [Dong95].

See the Appendix A Linear Solvers Basics for discussion of terms and concepts related to the ISS routines.

The term *RCI* indicates that when the solver needs the results of certain operations (for example, matrix-vector multiplications), the user performs them and passes the result to the solver. This makes the solver more universal as it is independent of the specific implementation of the operations like the matrix-vector multiplication. To perform such operations, the user can use the built-in sparse matrix-vector multiplications and triangular solvers routines described in Sparse BLAS Level 2 and Level 3 Routines.

> **NOTE**
> The RCI CG solver is implemented in two versions: for system of equations with a single right-hand side, and for systems of equations with multiple right-hand sides.
>
> The CG method may fail to compute the solution or compute the wrong solution if the matrix of the system is not symmetric and not positive definite.
>
> The FGMRES method may fail if the matrix is degenerate.

Table "RCI CG Interface Routines" lists the names of the routines, and describes their general use.

RCI ISS Interface Routines

Routine	Description
dcg_init, dcgmrhs_init, dfgmres_init	Initializes the solver.
dcg_check, dcgmrhs_check, dfgmres_check	Checks the consistency and correctness of the user defined data.
dcg, dcgmrhs, dfgmres	Computes the approximate solution vector.
dcg_get, dcgmrhs_get, dfgmres_get	Retrieves the number of the current iteration.

The Intel® oneAPI Math Kernel Library (oneMKL) RCI ISS interface routines are normally invoked in this order:

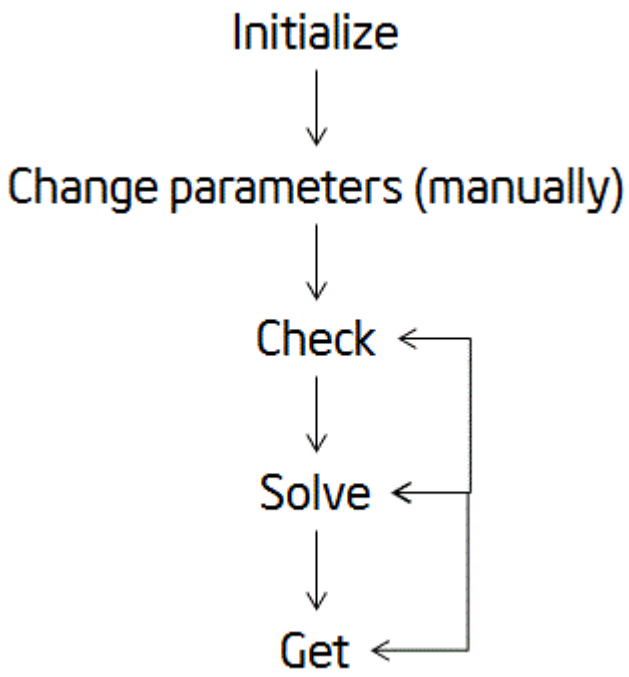
1. `<system_type>_init`
2. `<system_type>_check`
3. `<system_type>`
4. `<system_type>_get`

Advanced users can change that order if they need it. Others should follow the above order of calls.

The following diagram indicates the typical order in which the RCI ISS interface routines are invoked.

__border__top

Typical Order for Invoking RCI ISS interface Routines



See the code examples that use the RCI ISS interface routines to solve systems of linear equations in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory.

- `examples/solverf/source`

CG Interface Description

All types in this documentation refer to the common Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

Each routine for the RCI CG solver is implemented in two versions: for a system of equations with a single right-hand side (SRHS), and for a system of equations with multiple right-hand sides (MRHS). The names of routines for a system with MRHS contain the suffix `mrhs`.

Routine Options

All of the RCI CG routines have common parameters for passing various options to the routines (see CG Common Parameters). The values for these parameters can be changed during computations.

User Data Arrays

Many of the RCI CG routines take arrays of user data as input. For example, user arrays are passed to the routine dcgto compute the solution of a system of linear algebraic equations. The Intel® oneAPI Math Kernel Library (oneMKL) RCI CG routines do not make copies of the user input arrays to minimize storage requirements and improve overall run-time efficiency.

CG Common Parameters

NOTE
The default and initial values listed below are assigned to the parameters by calling the dcg_init/dcgmrhs_init routine.

n INTEGER, this parameter sets the size of the problem in the dcg_init/dcgmrhs_init routine. All the other routines use the *ipar*(1) parameter instead. Note that the coefficient matrix *A* is a square matrix of size *n***n*.

x DOUBLE PRECISION array of size *n* for SRHS, or matrix of size (*n***nrhs*) for MRHS. This parameter contains the current approximation to the solution. Before the first call to the dcg/dcgmrhs routine, it contains the initial approximation to the solution.

nrhs INTEGER, this parameter sets the number of right-hand sides for MRHS routines.

b DOUBLE PRECISION array containing a single right-hand side vector, or matrix of size *n*nrhs* containing right-hand side vectors.

RCI_request INTEGER, this parameter gives information about the result of work of the RCI CG routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions:

RCI_request= 1	multiply the matrix by *tmp* (1:*n*,1), put the result in *tmp*(1:*n*,2), and return the control to the dcg/dcgmrhs routine;
RCI_request= 2	to perform the stopping tests. If they fail, return the control to the dcg/dcgmrhs routine. If the stopping tests succeed, it indicates that the solution is found and stored in the *x* array;
RCI_request= 3	for SRHS: apply the preconditioner to *tmp*(1:*n*,3), put the result in *tmp*(1:*n*,4), and return the control to the dcg routine; for MRHS: apply the preconditioner to *tmp*(:,3+*ipar*(3)), put the result in *tmp*(:,3), and return the control to the dcgmrhs routine.

Note that the dcg_get/dcgmrhs_get routine does not change the parameter *RCI_request*. This enables use of this routine inside the *reverse communication* computations.

ipar INTEGER array, of size 128 for SRHS, and of size (128+2*nrhs) for MRHS. This parameter specifies the integer set of data for the RCI CG computations:

ipar(1) specifies the size of the problem. The dcg_init/dcgmrhs_init routine assigns *ipar*(1)=*n*. All the other routines use this parameter instead of *n*. There is no default value for this parameter.

ipar(2) specifies the type of output for error and warning messages generated by the RCI CG routines. The default value 6 means that all messages are displayed on the screen. Otherwise, the error and warning messages are written to the newly created files dcg_errors.txt and dcg_check_warnings.txt, respectively. Note that if *ipar*(6) and *ipar*(7) parameters are set to 0, error and warning messages are not generated at all.

ipar(3) for SRHS: contains the current stage of the RCI CG computations. The initial value is 1;

for MRHS: contains the number of the right-hand side for which the calculations are currently performed.

WARNING
Avoid altering this variable during computations.

ipar(4) contains the current iteration number. The initial value is 0.

ipar(5) specifies the maximum number of iterations. The default value is min(150, *n*).

ipar(6) if the value is not equal to 0, the routines output error messages in accordance with the parameter *ipar*(2). Otherwise, the routines do not output error messages at all, but return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(7) if the value is not equal to 0, the routines output warning messages in accordance with the parameter *ipar*(2). Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(8) if the value is not equal to 0, the dcg/dcgmrhs routine performs the stopping test for the maximum number of iterations: *ipar*(4) ≤

ipar(5). Otherwise, the method is stopped and the corresponding value is assigned to the *RCI_request*. If the value is 0, the routine does not perform this stopping test. The default value is 1.

ipar(9) — if the value is not equal to 0, the dcg/dcgmrhs routine performs the residual stopping test: *dpar*(5) ≤ *dpar*(4)= *dpar*(1)***dpar*(3)+ *dpar*(2). Otherwise, the method is stopped and corresponding value is assigned to the *RCI_request*. If the value is 0, the routine does not perform this stopping test. The default value is 0.

ipar(10) — if the value is not equal to 0, the dcg/dcgmrhs routine requests a user-defined stopping test by setting the output parameter *RCI_request*=2. If the value is 0, the routine does not perform the user defined stopping test. The default value is 1.

NOTE
At least one of the parameters *ipar*(8)-*ipar*(10) must be set to 1.

ipar(11) — if the value is equal to 0, the dcg/dcgmrhs routine runs the non-preconditioned version of the corresponding CG method. Otherwise, the routine runs the preconditioned version of the CG method, and by setting the output parameter *RCI_request*=3, indicates that you must perform the preconditioning step. The default value is 0.

ipar(12:128) — are reserved and not used in the current RCI CG SRHS and MRHS routines.

NOTE
For future compatibility, you must declare the array *ipar* with length 128 for a single right-hand side.

ipar(12:128+2**nrhs*) — are reserved for internal use in the current RCI CG SRHS and MRHS routines.

NOTE
For future compatibility, you must declare the array *ipar* with length 128+2**nrhs* for multiple right-hand sides.

dpar

DOUBLE PRECISION array, for SRHS of size 128, for MRHS of size (128+2*nrhs); this parameter is used to specify the double precision set of data for the RCI CG computations, specifically:

dpar(1)	specifies the relative tolerance. The default value is $1.0X10^{-6}$.
dpar(2)	specifies the absolute tolerance. The default value is 0.0.
dpar(3)	specifies the square norm of the initial residual (if it is computed in the dcg/dcgmrhs routine). The initial value is 0.0.
dpar(4)	service variable equal to *dpar*(1)*dpar(3)+*dpar*(2) (if it is computed in the dcg/dcgmrhs routine). The initial value is 0.0.
dpar(5)	specifies the square norm of the current residual. The initial value is 0.0.
dpar(6)	specifies the square norm of residual from the previous iteration step (if available). The initial value is 0.0.
dpar(7)	contains the *alpha* parameter of the CG method. The initial value is 0.0.
dpar(8)	contains the *beta* parameter of the CG method, it is equal to *dpar*(5)/*dpar*(6) The initial value is 0.0.
dpar(9:128)	are reserved and not used in the current RCI CG SRHS and MRHS routines.

NOTE
For future compatibility, you must declare the array *dpar* with length 128 for a single right-hand side.

dpar(9:128+2*nrhs) (9:128 + 2*nrhs)	are reserved for internal use in the current RCI CG SRHS and MRHS routines.

NOTE
For future compatibility, you must declare the array *dpar* with length 128+2*nrhs for multiple right-hand sides.

tmp

DOUBLE PRECISION array of size (n, 4) for SRHS, and (n, (3+nrhs)) for MRHS. This parameter is used to supply the double precision temporary space for the RCI CG computations, specifically:

tmp(:,1)	specifies the current search direction. The initial value is 0.0.
tmp(:,2)	contains the matrix multiplied by the current search direction. The initial value is 0.0.
tmp(:,3)	contains the current residual. The initial value is 0.0.
tmp(:,4)	contains the inverse of the preconditioner applied to the current residual for the SRHS version of CG. There is no initial value for this parameter.
tmp(:,4:3+*nrhs*)	contains the inverse of the preconditioner applied to the current residual for the MRHS version of CG. There is no initial value for this parameter.

NOTE
You can define this array in the code using RCI CG SRHS as DOUBLE PRECISION*tmp*(*n*,3) if you run only non-preconditioned CG iterations.

Schemes of Using the RCI CG Routines

The following pseudocode shows the general schemes of using the RCI CG routines for the SRHS case. The MRHS is similar (see the example code for more details).

```
...
generate matrix A
generate preconditioner C (optional)
   call dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)
   change parameters in ipar, dpar if necessary
   call dcg_check(n, x, b, RCI_request, ipar, dpar, tmp)
1  call dcg(n, x, b, RCI_request, ipar, dpar, tmp)
   if (RCI_request.eq.1) then
    multiply the matrix A by tmp(1:n,1) and put the result in tmp(1:n,2)
   It is possible to use MKL Sparse BLAS Level 2 subroutines for this operation
c  proceed with CG iterations
     goto 1
   endif
   if (RCI_request.eq.2)then
    do the stopping test
    if (test not passed) then
c  proceed with CG iterations
```

```
     go to 1
    else
c  stop CG iterations
     goto 2
    endif
   endif
   if (RCI_request.eq.3) then (optional)
   apply the preconditioner C inverse to tmp(1:n,3) and put the result in tmp(1:n,4)
c  proceed with CG iterations
    goto 1
   end
```

2 call dcg_get(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*, *itercount*)

current iteration number is in *itercount*

the computed approximation is in the array *x*

FGMRES Interface Description

All types in this documentation refer to the common Fortran types: `INTEGER` and `DOUBLE PRECISION`.

Routine Options

All of the RCI FGMRES routines have common parameters for passing various options to the routines (see FGMRES Common Parameters). The values for these parameters can be changed during computations.

User Data Arrays

Many of the RCI FGMRES routines take arrays of user data as input. For example, user arrays are passed to the routine `dfgmres`to compute the solution of a system of linear algebraic equations. To minimize storage requirements and improve overall run-time efficiency, the Intel® oneAPI Math Kernel Library (oneMKL) RCI FGMRES routines do not make copies of the user input arrays.

FGMRES Common Parameters

NOTE
The default and initial values listed below are assigned to the parameters by calling the `dfgmres_init` routine.

n — `INTEGER`, this parameter sets the size of the problem in the `dfgmres_init` routine. All the other routines use the *ipar*(1) parameter instead. Note that the coefficient matrix *A* is a square matrix of size *n***n*.

x — `DOUBLE PRECISION` array, this parameter contains the current approximation to the solution vector. Before the first call to the `dfgmres` routine, it contains the initial approximation to the solution vector.

b

DOUBLE PRECISION array, this parameter contains the right-hand side vector. Depending on user requests (see the parameter *ipar*(13)), it might contain the approximate solution after execution.

RCI_request

INTEGER, this parameter gives information about the result of work of the RCI FGMRES routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions:

RCI_request= 1	multiply the matrix by *tmp*(*ipar*(22)), put the result in *tmp*(*ipar*(23)), and return the control to the dfgmres routine;
RCI_request= 2	perform the stopping tests. If they fail, return the control to the dfgres routine. Otherwise, the solution can be updated by a subsequent call to dfgmres_get routine;
RCI_request= 3	apply the preconditioner to *tmp*(*ipar*(22)), put the result in *tmp*(*ipar*(23)), and return the control to the dfgmres routine.
RCI_request= 4	check if the norm of the current orthogonal vector is zero, within the rounding or computational errors. Return the control to the dfgmres routine if it is not zero, otherwise complete the solution process by calling dfgmres_get routine.

ipar(128)[128]

INTEGER array, this parameter specifies the integer set of data for the RCI FGMRES computations:

ipar(1)	specifies the size of the problem. The dfgmres_init routine assigns *ipar*(1)=*n*. All the other routines uses this parameter instead of *n*. There is no default value for this parameter.
ipar(2)	specifies the type of output for error and warning messages that are generated by the RCI FGMRES routines. The default value 6 means that all messages are displayed on the screen. Otherwise the error and warning messages are written to the newly created file MKL_RCI_FGMRES_Log.txt. Note that if *ipar*(6) and *ipar*(7) parameters are set to 0, error and warning messages are not generated at all.
ipar(3)	contains the current stage of the RCI FGMRES computations. The initial value is 1.

WARNING
Avoid altering this variable during computations.

ipar(4) contains the current iteration number. The initial value is 0.

ipar(5) specifies the maximum number of iterations. The default value is *min* (150,*n*).

ipar(6) if the value is not 0, the routines output error messages in accordance with the parameter *ipar*(2). If it is 0, the routines do not output error messages at all, but return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(7) if the value is not 0, the routines output warning messages in accordance with the parameter *ipar*(2). Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(8) if the value is not equal to 0, the dfgmres routine performs the stopping test for the maximum number of iterations: *ipar*(4) ≤*ipar*(5). If the value is 0, the dfgmres routine does not perform this stopping test. The default value is 1.

ipar(9) if the value is not 0, the dfgmres routine performs the residual stopping test: *dpar*(5) ≤*dpar*(4).If the value is 0, the dfgmres routine does not perform this stopping test. The default value is 0.

ipar(10) if the value is not 0, the dfgmres routine indicates that the user-defined stopping test should be performed by setting *RCI_request*=2. If the value is 0, the dfgmres routine does not perform the user-defined stopping test. The default value is 1.

NOTE
At least one of the parameters *ipar*(8)-*ipar*(10) must be set to 1.

ipar(11) if the value is 0, the dfgmres routine runs the non-preconditioned version of the FGMRES method. Otherwise, the routine runs the preconditioned version of the FGMRES method,

and requests that you perform the preconditioning step by setting the output parameter *RCI_request*=3. The default value is 0.

ipar(12) if the value is not equal to 0, the dfgmres routine performs the automatic test for zero norm of the currently generated vector: *dpar*(7)≤*dpar*(8), where *dpar*(8) contains the tolerance value. Otherwise, the routine indicates that you must perform this check by setting the output parameter *RCI_request*=4. The default value is 0.

ipar(13) if the value is equal to 0, the dfgmres_get routine updates the solution to the vector *x* according to the computations done by the dfgmres routine. If the value is positive, the routine writes the solution to the right-hand side vector *b*. If the value is negative, the routine returns only the number of the current iteration, and does not update the solution. The default value is 0.

NOTE
It is possible to call the dfgmres_get routine at any place in the code, but you must pay special attention to the parameter *ipar*(13). The RCI FGMRES iterations can be continued after the call to dfgmres_get routine only if the parameter *ipar*(13) is not equal to zero. If *ipar*(13) is positive, then the updated solution overwrites the right-hand side in the vector *b*. If you want to run the restarted version of FGMRES with the same right-hand side, then it must be saved in a different memory location before the first call to the dfgmres_get routine with positive *ipar*(13).

ipar(14) contains the internal iteration counter that counts the number of iterations before the restart takes place. The initial value is 0.

WARNING
Do not alter this variable during computations.

ipar(15) specifies the number of the non-restarted FGMRES iterations. To run the restarted version of the FGMRES method, assign the number of

iterations to *ipar*(15) before the restart. The default value is *min*(150, *n*), which means that by default the non-restarted version of FGMRES method is used.

ipar(16) — service variable specifying the location of the rotated Hessenberg matrix from which the matrix stored in the packed format (see Matrix Arguments in the Appendix B for details) is started in the *tmp* array.

ipar(17) — service variable specifying the location of the rotation cosines from which the vector of cosines is started in the *tmp* array.

ipar(18) — service variable specifying the location of the rotation sines from which the vector of sines is started in the *tmp* array.

ipar(19) — service variable specifying the location of the rotated residual vector from which the vector is started in the *tmp* array.

ipar(20) — service variable, specifies the location of the least squares solution vector from which the vector is started in the *tmp* array.

ipar(21) — service variable specifying the location of the set of preconditioned vectors from which the set is started in the *tmp* array. The memory locations in the *tmp* array starting from *ipar*(21) are used only for the preconditioned FGMRES method.

ipar(22) — specifies the memory location from which the first vector (source) used in operations requested via *RCI_request* is started in the *tmp* array.

ipar(23) — specifies the memory location from which the second vector (output) used in operations requested via *RCI_request* is started in the *tmp* array.

ipar(24:128) — are reserved and not used in the current RCI FGMRES routines.

NOTE
You must declare the array *ipar* with length 128. While defining the array in the code as INTEGER*ipar*(23) works, there is no guarantee of future compatibility with Intel® oneAPI Math Kernel Library (oneMKL).

dpar(128)

DOUBLE PRECISION array, this parameter specifies the double precision set of data for the RCI CG computations, specifically:

dpar(1)	specifies the relative tolerance. The default value is 1.0e-6.
dpar(2)	specifies the absolute tolerance. The default value is 0.0e-0.
dpar(3)	specifies the Euclidean norm of the initial residual (if it is computed in the dfgmres routine). The initial value is 0.0.
dpar(4)	service variable equal to *dpar*(1)**dpar*(3)+*dpar*(2) (if it is computed in the dfgmres routine). The initial value is 0.0.
dpar(5)	specifies the Euclidean norm of the current residual. The initial value is 0.0.
dpar(6)	specifies the Euclidean norm of residual from the previous iteration step (if available). The initial value is 0.0.
dpar(7)	contains the norm of the generated vector. The initial value is 0.0.

NOTE
In terms of [Saad03] this parameter is the coefficient $h_{k+1,k}$ of the Hessenberg matrix.

dpar(8)	contains the tolerance for the zero norm of the currently generated vector. The default value is 1.0e-12.
dpar(9:128)	are reserved and not used in the current RCI FGMRES routines.

NOTE
You must declare the array *dpar* with length 128. While defining the array in the code as DOUBLE PRECISION *dpar*(8) works, there is no guarantee of future compatibility with Intel® oneAPI Math Kernel Library (oneMKL).

tmp

DOUBLE PRECISION array of size ((2**ipar*(15)+1)**n* + *ipar*(15)*(*ipar*(15)+9)/2 + 1)) used to supply the double precision temporary space for the RCI FGMRES computations, specifically:

tmp(1:ipar(16)*-1)*	contains the sequence of vectors generated by the FGMRES method. The initial value is 0.0.

tmp(ipar(16):ipar(17)-1)	contains the rotated Hessenberg matrix generated by the FGMRES method; the matrix is stored in the packed format. There is no initial value for this part of *tmp* array.
tmp(ipar(17):ipar(18)-1)	contains the rotation cosines vector generated by the FGMRES method. There is no initial value for this part of *tmp* array.
tmp(ipar(18):ipar(19)-1)	contains the rotation sines vector generated by the FGMRES method. There is no initial value for this part of *tmp* array.
tmp(ipar(19):ipar(20)-1)	contains the rotated residual vector generated by the FGMRES method. There is no initial value for this part of *tmp* array.
tmp(ipar(20):ipar(21)-1)	contains the solution vector to the least squares problem generated by the FGMRES method. There is no initial value for this part of *tmp* array.
tmp(ipar(21):)	contains the set of preconditioned vectors generated for the FGMRES method by the user. This part of *tmp* array is not used if the non-preconditioned version of FGMRES method is called. There is no initial value for this part of *tmp* array.

NOTE
You can define this array in the code as `DOUBLE PRECISION tmp((2*ipar(15)+1)*n + ipar(15)*(ipar(15)+9)/2 + 1))` if you run only non-preconditioned FGMRES iterations.

Scheme of Using the RCI FGMRES Routines

The following pseudocode shows the general scheme of using the RCI FGMRES routines.

```
...
generate matrix A
generate preconditioner C (optional)
  call dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
   change parameters in ipar, dpar if necessary
   call dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
1  call dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
   if (RCI_request.eq.1) then
     multiply the matrix A by tmp(ipar(22)) and put the result in tmp(ipar(23))
```

It is possible to use MKL Sparse BLAS Level 2 subroutines for this operation

```
c  proceed with FGMRES iterations
   goto 1
  endif
  if (RCI_request.eq.2) then
   do the stopping test
   if (test not passed) then
c  proceed with FGMRES iterations
    go to 1
   else
c  stop FGMRES iterations
    goto 2
   endif
  endif
  if (RCI_request.eq.3) then (optional)
  apply the preconditioner C inverse to tmp(ipar(22)) and put the result in tmp(ipar(23))
c  proceed with FGMRES iterations
   goto 1
  endif
  if (RCI_request.eq.4) then
  check the norm of the next orthogonal vector, it is contained in dpar(7)
   if (the norm is not zero up to rounding/computational errors) then
c  proceed with FGMRES iterations
    goto 1
   else
c  stop FGMRES iterations
    goto 2
   endif
  endif
```

2 call `dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)`

current iteration number is in *itercount*

the computed approximation is in the array *x*

NOTE

For the FGMRES method, the array *x* initially contains the current approximation to the solution. It can be updated only by calling the routine `dfgmres_get`, which updates the solution in accordance with the computations performed by the routine `dfgmres`.

The above pseudocode demonstrates two main differences in the use of RCI FGMRES interface comparing with the CG Interface Description. The first difference relates to *RCI_request*=3: it uses different locations in the *tmp* array, which is two-dimensional for CG and one-dimensional for FGMRES. The second difference relates to *RCI_request*=4: the RCI CG interface never produces *RCI_request*=4.

RCI ISS Routines

dcg_init

Initializes the solver.

Syntax

`dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcg_init`initializes the solver. After initialization, all subsequent invocations of the Intel® oneAPI Math Kernel Library (oneMKL) RCI CG routines use the values of all parameters returned by the routine`dcg_init`. Advanced users can skip this step and set the values in the *ipar* and *dpar* arrays directly.

> **Caution**
> You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dcg_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

n	`INTEGER`. Sets the size of the problem.
x	`DOUBLE PRECISION`. Array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.
b	`DOUBLE PRECISION`. Array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request	`INTEGER`. Gives information about the result of the routine.
ipar	`INTEGER` . Array of size `128`. Refer to the CG Common Parameters.
dpar	`DOUBLE PRECISION`. Array of size `128`. Refer to the CG Common Parameters.
tmp	`DOUBLE PRECISION` . Array of size `(n,4)`. Refer to the CG Common Parameters.

Return Values

RCI_request= 0	Indicates that the task completed normally.

RCI_request= -10000	Indicates failure to complete the task.

dcg_check

Checks consistency and correctness of the user defined data.

Syntax

`dcg_check(n, x, b, RCI_request, ipar, dpar, tmp)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcg_check` checks consistency and correctness of the parameters to be passed to the solver routine `dcg`. However this operation does not guarantee that the solver returns the correct result. It only reduces the chance of making a mistake in the parameters of the method. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors must be defined in a previous call to the `dcg_init` routine.

If none of the stopping criteria (*ipar*(8)-*ipar*(10)) has been enabled, both *ipar*(8) and *ipar*(9) will be set to 1.

Input Parameters

ipar	Array of size 128. Refer to the FGMRES Common Parameters.
n	`INTEGER`. Sets the size of the problem.
x	`DOUBLE PRECISION`. Array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.
b	`DOUBLE PRECISION`. Array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request	`INTEGER`. Gives information about result of the routine.
ipar	`INTEGER`. Array of size `128`. Refer to the CG Common Parameters. Only *ipar*(8)-*ipar*(9) might be changed
dpar	`DOUBLE PRECISION`. Array of size `128`. Refer to the CG Common Parameters.
tmp	`DOUBLE PRECISION`. Array of size `(n,4)`. Refer to the CG Common Parameters.

Return Values

RCI_request= 0	Indicates that the task completed normally.
RCI_request= -1100	Indicates that the task is interrupted and the errors occur.
RCI_request= -1001	Indicates that there are some warning messages.

RCI_request= -1010	Indicates that the routine changed some parameters to make them consistent or correct.
RCI_request= -1011	Indicates that there are some warning messages and that the routine changed some parameters.

dcg

Computes the approximate solution vector.

Syntax

`dcg(n, x, b, RCI_request, ipar, dpar, tmp)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `dcg` routine computes the approximate solution vector using the CG method [Young71]. The routine `dcg` uses the vector in the array *x* before the first call as an initial approximation to the solution. The parameter *RCI_request* gives you information about the task completion and requests results of certain operations that are required by the solver.

Note that lengths of all vectors must be defined in a previous call to the `dcg_init` routine.

Input Parameters

n	`INTEGER`. Sets the size of the problem.
x	`DOUBLE PRECISION` . Array of size *n*. Contains the initial approximation to the solution vector.
b	`DOUBLE PRECISION` . Array of size *n*. Contains the right-hand side vector.
tmp	`DOUBLE PRECISION` . Array of size `(`*n*, 4). Refer to the CG Common Parameters.

Output Parameters

RCI_request	`INTEGER`. Gives information about result of work of the routine.
x	`DOUBLE PRECISION` . Array of size *n*. Contains the updated approximation to the solution vector.
ipar	`INTEGER` . Array of size *128*. Refer to the CG Common Parameters.
dpar	`DOUBLE PRECISION` . Array of size *128*. Refer to the CG Common Parameters.
tmp	`DOUBLE PRECISION` . Array of size `(`*n*,4). Refer to the CG Common Parameters.

Return Values

RCI_request=0	Indicates that the task completed normally and the solution is found and stored in the vector *x*. This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the *RCI_request*= 2.
RCI_request=-1	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.
RCI_request=-2	Indicates that the routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.
RCI_request=- 10	Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value *dpar*(6) was altered outside of the routine, or the dcg_check routine was not called.
RCI_request=-11	Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values *ipar*(8), *ipar*(9), *ipar*(10) were altered outside of the routine, or the dcg_check routine was not called.
RCI_request= 1	Indicates that you must multiply the matrix by *tmp*(1:*n*,1), put the result in the *tmp*(1:*n*,2), and return control back to the routine dcg.
RCI_request= 2	Indicates that you must perform the stopping tests. If they fail, return control back to the dcg routine. Otherwise, the solution is found and stored in the vector *x*.
RCI_request= 3	Indicates that you must apply the preconditioner to *tmp*(:, 3), put the result in the *tmp*(:, 4), and return control back to the routine dcg.

dcg_get

Retrieves the number of the current iteration.

Syntax

dcg_get(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*, *itercount*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The routine dcg_get retrieves the current iteration number of the solutions process.

Input Parameters

n	INTEGER. Sets the size of the problem.

x	DOUBLE PRECISION. Array of size *n*. Contains the approximation vector to the solution.
b	DOUBLE PRECISION. Array of size *n*. Contains the right-hand side vector.
RCI_request	INTEGER. This parameter is not used.
ipar	INTEGER. Array of size 128. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION. Array of size 128. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION. Array of size (*n*,4). Refer to the CG Common Parameters.

Output Parameters

itercount	INTEGER. Returns the current iteration number.

Return Values

The routine dcg_get has no return values.

dcgmrhs_init

Initializes the RCI CG solver with MHRS.

Syntax

dcgmrhs_init(*n*, *x*, *nrhs*, *b*, *method*, *RCI_request*, *ipar*, *dpar*, *tmp*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The routine dcgmrhs_initinitializes the solver. After initialization all subsequent invocations of the Intel® oneAPI Math Kernel Library (oneMKL) RCI CG with multiple right-hand sides (MRHS) routines use the values of all parameters that are returned bydcgmrhs_init. Advanced users may skip this step and set the values to these parameters directly in the appropriate routines.

WARNING
You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the dcgmrhs_check routine, but it does not guarantee that the method will work correctly.

Input Parameters

n	INTEGER. Sets the size of the problem.
x	DOUBLE PRECISION. Array of size *n*nrhs*. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to *b*.
nrhs	INTEGER. Sets the number of right-hand sides.

b	DOUBLE PRECISION. Array of size *n*nrhs*. Contains the right-hand side vectors.
method	INTEGER. Specifies the method of solution: A value of 1 indicates CG with multiple right-hand sides (default value)

Output Parameters

RCI_request	INTEGER. Gives information about the result of the routine.
ipar	INTEGER. Array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION. Array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION. Array of size (*n*, (3+*nrhs*)). Refer to the CG Common Parameters.

Return Values

RCI_request= 0	Indicates that the task completed normally.
RCI_request= -10000	Indicates failure to complete the task.

dcgmrhs_check

Checks consistency and correctness of the user defined data.

Syntax

dcgmrhs_check(*n*, *x*, *nrhs*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The routine dcgmrhs_check checks the consistency and correctness of the parameters to be passed to the solver routine dcgmrhs. While this operation reduces the chance of making a mistake in the parameters, it does not guarantee that the solver returns the correct result.

If you are sure that the correct data is *specified* in the solver parameters, you can skip this operation.

The lengths of all vectors must be defined in a previous call to the dcgmrhs_init routine.

If none of the stopping criteria (*ipar*(8)-*ipar*(10)) has been enabled, both *ipar*(8) and *ipar*(9) will be set to 1.

Input Parameters

n	INTEGER. Sets the size of the problem.
x	DOUBLE PRECISION. Array of size *n*nrhs*. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to *b*.
nrhs	INTEGER. This parameter sets the number of right-hand sides.

b	DOUBLE PRECISION. Array of size *n*nrhs*. Contains the right-hand side vectors.

Output Parameters

RCI_request	INTEGER. Returns information about the results of the routine.
ipar	INTEGER. Array of size *(128+2*nrhs)*. Refer to the CG Common Parameters. Only *ipar*(8)-*ipar*(9) might be changed.
dpar	DOUBLE PRECISION. Array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION. Array of size (*n*, (3+*nrhs*)). Refer to the CG Common Parameters.

Return Values

RCI_request= 0	Indicates that the task completed normally.
RCI_request= -1100	Indicates that the task is interrupted and the errors occur.
RCI_request= -1001	Indicates that there are some warning messages.
RCI_request= -1010	Indicates that the routine changed some parameters to make them consistent or correct.
RCI_request= -1011	Indicates that there are some warning messages and that the routine changed some parameters.

dcgmrhs

Computes the approximate solution vectors.

Syntax

dcgmrhs(*n*, *x*, *nrhs*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The routine dcgmrhs computes approximate solution vectors using the CG with multiple right-hand sides (MRHS) method [Young71]. The routine dcgmrhs uses the value that was in the *x* before the first call as an initial approximation to the solution. The parameter *RCI_request* gives information about task completion status and requests results of certain operations that are required by the solver.

Note that lengths of all vectors are assumed to have been defined in a previous call to the dcgmrhs_init routine.

Input Parameters

n	INTEGER. Sets the size of the problem, and the sizes of arrays *x* and *b*.
x	DOUBLE PRECISION. Array of size *n*nrhs*. Contains the initial approximation to the solution vectors.

nrhs	INTEGER. Sets the number of right-hand sides.
b	DOUBLE PRECISION. Array of size *n*nrhs*. Contains the right-hand side vectors.
tmp	DOUBLE PRECISION. Array of size (*n*, 3+*nrhs*). Refer to the CG Common Parameters.

Output Parameters

RCI_request	INTEGER. Gives information about result of work of the routine.
x	DOUBLE PRECISION. Array of size (*n* by *nrhs*). Contains the updated approximation to the solution vectors.
ipar	INTEGER. Array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION. Array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION. Array of size (*n*, (3+*nrhs*)). Refer to the CG Common Parameters.

Return Values

RCI_request=0	Indicates that the task completed normally and the solution is found and stored in the vector *x*. This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the *RCI_request*= 2.
RCI_request=-1	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if both tests are requested by the user.
RCI_request=-2	The routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.
RCI_request=- 10	Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value *dpar*(6) was altered outside of the routine, or the dcg_check routine was not called.
RCI_request=-11	Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values *ipar*(8), *ipar*(9), *ipar*(10) were altered outside of the routine, or the dcg_check routine was not called.
RCI_request= 1	Indicates that you must multiply the matrix by *tmp*(1:*n*,1), put the result in the *tmp*(1:*n*,2), and return control back to the routine dcg.
RCI_request= 2	Indicates that you must perform the stopping tests. If they fail, return control back to the dcg routine. Otherwise, the solution is found and stored in the vector *x*.

RCI_request= 3	Indicates that you must apply the preconditioner to *tmp*(:, 3), put the result in the *tmp*(:, 4), and return control back to the routine dcg.

dcgmrhs_get

Retrieves the number of the current iteration.

Syntax

dcgmrhs_get(*n*, *x*, *nrhs*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*, *itercount*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The routine dcgmrhs_get retrieves the current iteration number of the solving process.

Input Parameters

n	INTEGER. Sets the size of the problem.
x	DOUBLE PRECISION. Array of size *n*nrhs*. Contains the initial approximation to the solution vectors.
nrhs	INTEGER. Sets the number of right-hand sides.
b	DOUBLE PRECISION. Array of size *n*nrhs*. Contains the right-hand side .
RCI_request	INTEGER. This parameter is not used.
ipar	INTEGER. Array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION. Array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION. Array of size (*n*, (3+*nrhs*)). Refer to the CG Common Parameters.

Output Parameters

itercount	INTEGER. Array of size *nrhs*. Returns the current iteration number for each right-hand side.

Return Values

The routine dcgmrhs_get has no return values.

dfgmres_init

Initializes the solver.

Syntax

dfgmres_init(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*)

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dfgmres_init`initializes the solver. After initialization all subsequent invocations of Intel® oneAPI Math Kernel Library (oneMKL) RCI FGMRES routines use the values of all parameters that are returned by`dfgmres_init`. Advanced users can skip this step and set the values in the *ipar* and *dpar* arrays directly.

> **WARNING**
> You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dfgmres_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

n	`INTEGER`. Sets the size of the problem.
x	`DOUBLE PRECISION`. Array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.
b	`DOUBLE PRECISION`. Array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request	`INTEGER`. Gives information about the result of the routine.
ipar	`INTEGER`. Array of size `128`. Refer to the FGMRES Common Parameters.
dpar	`DOUBLE PRECISION`. Array of size `128`. Refer to the FGMRES Common Parameters.
tmp	`DOUBLE PRECISION`. Array of size `((2*`*ipar*`(15) + 1)*`*n* `+` *ipar*`(15)*(`*ipar*`(15) + 9)/2 + 1)`. Refer to the FGMRES Common Parameters.

Return Values

RCI_request= 0	Indicates that the task completed normally.
RCI_request= -10000	Indicates failure to complete the task.

dfgmres_check

Checks consistency and correctness of the user defined data.

Syntax

`dfgmres_check(`*n*`, `*x*`, `*b*`, `*RCI_request*`, `*ipar*`, `*dpar*`, `*tmp*`)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine dfgmres_check checks consistency and correctness of the parameters to be passed to the solver routine dfgmres. However, this operation does not guarantee that the method gives the correct result. It only reduces the chance of making a mistake in the parameters of the routine. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors are assumed to have been defined in a previous call to the dfgmres_init routine.

In particular, the routine checks the consistency of *ipar*(16)-*ipar*(21) and *ipar*(1), *ipar*(15). If the values do not agree, the routine emits a warning and modifies *ipar*(16)-*ipar*(21) to comply with the values of *ipar*(1), *ipar*(15). A possible use case for this modification is a non-default value (not the one set by a possible call to dfgmres_init) of *ipar*(15).

Also, if none of the stopping criteria (*ipar*(8)-*ipar*(10)) has been enabled, both *ipar*(8) and *ipar*(9) will be set to 1.

NOTE: It is not strictly necessary to call the dfgmres_check routine unless the values of *ipar*(15) or *ipar*(1) are changed after the last call to dfgmres_init.

Input Parameters

ipar	Array of size 128. Refer to the FGMRES Common Parameters.
n	INTEGER. Sets the size of the problem.
x	DOUBLE PRECISION. Array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.
b	DOUBLE PRECISION. Array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request	INTEGER. Gives information about result of the routine.
ipar	INTEGER. Array of size *128*. Refer to the FGMRES Common Parameters. Only *ipar*(8)-*ipar*(9) and *ipar*(16)-*ipar*(21) might be changed.
dpar	DOUBLE PRECISION. Array of size *128*. Refer to the FGMRES Common Parameters.
tmp	DOUBLE PRECISION. Array of size ((2**ipar*(15) + 1)**n* + *ipar*(15)*(*ipar*(15) + 9)/2 + 1). Refer to the FGMRES Common Parameters.

Return Values

RCI_request= 0	Indicates that the task completed normally.
RCI_request= -1100	Indicates that the task is interrupted and the errors occur.
RCI_request= -1001	Indicates that there are some warning messages.
RCI_request= -1010	Indicates that the routine changed some parameters to make them consistent or correct.
RCI_request= -1011	Indicates that there are some warning messages and that the routine changed some parameters.

dfgmres

Makes the FGMRES iterations.

Syntax

`dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dfgmres` performs the FGMRES iterations [Saad03], using the value that was in the array `x` before the first call as an initial approximation of the solution vector. To update the current approximation to the solution, the `dfgmres_get` routine must be called. The RCI FGMRES iterations can be continued after the call to the `dfgmres_get` routine only if the value of the parameter `ipar(13)` is not equal to 0 (default value). Note that the updated solution overwrites the right-hand side in the vector *b* if the parameter `ipar(13)` is positive, and the restarted version of the FGMRES method can not be run. If you want to keep the right-hand side, you must be save it in a different memory location before the first call to the `dfgmres_get` routine with a positive `ipar(13)`.

The parameter `RCI_request` gives information about the task completion and requests results of certain operations that the solver requires.

The lengths of all the vectors must be defined in a previous call to the `dfgmres_init` routine.

Input Parameters

`n`	`INTEGER`. Sets the size of the problem.
`x`	`DOUBLE PRECISION`. Array of size *n*. Contains the initial approximation to the solution vector.
`b`	`DOUBLE PRECISION`. Array of size *n*. Contains the right-hand side vector.
`tmp`	`DOUBLE PRECISION`. Array of size `(13)`. Refer to the FGMRES Common Parameters.

Output Parameters

`RCI_request`	`INTEGER`. Informs about result of work of the routine.
`ipar`	`INTEGER`. Array of size `128`. Refer to the FGMRES Common Parameters.
`dpar`	`DOUBLE PRECISION`. Array of size `128`. Refer to the FGMRES Common Parameters.
`tmp`	`DOUBLE PRECISION`. Array of size `((2*ipar(15)+1)*n +ipar(15)*ipar(15)+9)/2 + 1`. Refer to the FGMRES Common Parameters.

Return Values

RCI_request=0	Indicates that the task completed normally and the solution is found and stored in the vector *x*. This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the *RCI_request*= 2 or 4.
RCI_request=-1	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.
RCI_request= -10	Indicates that the routine was interrupted because of an attempt to divide by zero. Usually this happens if the matrix is degenerate or almost degenerate. However, it may happen if the parameter *dpar* is altered, or if the method is not stopped when the solution is found.
RCI_request= -11	Indicates that the routine was interrupted because it entered an infinite cycle. Usually this happens because the values *ipar*(8), *ipar*(9), *ipar*(10) were altered outside of the routine, or the dfgmres_check routine was not called.
RCI_request= -12	Indicates that the routine was interrupted because errors were found in the method parameters. Usually this happens if the parameters *ipar* and *dpar* were altered by mistake outside the routine.
RCI_request= 1	Indicates that you must multiply the matrix by *tmp*(*ipar*(22)), put the result in the *tmp*(*ipar*(23)), and return control back to the routine dfgmres.
RCI_request= 2	Indicates that you must perform the stopping tests. If they fail, return control to the dfgmres routine. Otherwise, the FGMRES solution is found, and you can run the fgmres_get routine to update the computed solution in the vector *x*.
RCI_request= 3	Indicates that you must apply the inverse preconditioner to *ipar*(22), put the result in the *ipar*(23), and return control back to the routine dfgmres.
RCI_request= 4	Indicates that you must check the norm of the currently generated vector. If it is not zero within the computational/ rounding errors, return control to the dfgmres routine. Otherwise, the FGMRES solution is found, and you can run the dfgmres_get routine to update the computed solution in the vector *x*.

dfgmres_get

Retrieves the number of the current iteration and updates the solution.

Syntax

`dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dfgmres_get` retrieves the current iteration number of the solution process and updates the solution according to the computations performed by the `dfgmres` routine. To retrieve the current iteration number only, set the parameter `ipar(13)` = -1 beforehand. Normally, you should do this before proceeding further with the computations. If the intermediate solution is needed, the method parameters must be set properly. For details see FGMRES Common Parametersand the Iterative Sparse Solver code examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- `examples/solverf/source`

Input Parameters

n	`INTEGER`. Sets the size of the problem.
ipar	`INTEGER`. Array of size *128*. Refer to the FGMRES Common Parameters.
dpar	`DOUBLE PRECISION`. Array of size *128*. Refer to the FGMRES Common Parameters.
tmp	`DOUBLE PRECISION`. Array of size `((2*ipar(15)+1)*n +ipar(15)*ipar(15)+9)/2 + 1)`. Refer to the FGMRES Common Parameters.

Output Parameters

x	`DOUBLE PRECISION`. Array of size *n*. If `ipar(13)` = 0, it contains the updated approximation to the solution according to the computations done in `dfgmres` routine. Otherwise, it is not changed.
b	`DOUBLE PRECISION`. Array of size *n*. If `ipar(13)` > 0, it contains the updated approximation to the solution according to the computations done in `dfgmres` routine. Otherwise, it is not changed.
RCI_request	`INTEGER`. Gives information about result of the routine.
itercount	`INTEGER`. Contains the value of the current iteration number.

Return Values

RCI_request = 0	Indicates that the task completed normally.
RCI_request = -12	Indicates that the routine was interrupted because errors were found in the routine parameters. Usually this happens if the parameters *ipar* and *dpar* were altered by mistake outside of the routine.
RCI_request = -10000	Indicates that the routine failed to complete the task.

RCI ISS Implementation Details

Several aspects of the Intel® oneAPI Math Kernel Library (oneMKL) RCI ISS interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, include one of the Intel® oneAPI Math Kernel Library (oneMKL) RCI ISS language-specific header files.

NOTE
Intel® oneAPI Math Kernel Library (oneMKL) does not support the RCI ISS interface unless you include the language-specific header file.

Preconditioners based on Incomplete LU Factorization Technique

Preconditioners, or accelerators are used to accelerate an iterative solution process. In some cases, their use can reduce the number of iterations dramatically and thus lead to better solver performance. Although the terms *preconditioner* and *accelerator* are synonyms, hereafter only *preconditioner* is used.

Intel® oneAPI Math Kernel Library (oneMKL) provides two preconditioners, ILU0 and ILUT, for sparse matrices presented in the format accepted in the Intel® oneAPI Math Kernel Library (oneMKL) direct sparse solvers (three-array variation of the CSR storage format described inSparse Matrix Storage Format). The algorithms used are described in [Saad03].

The ILU0 preconditioner is based on a well-known factorization of the original matrix into a product of two triangular matrices: lower and upper triangular matrices. Usually, such decomposition leads to some fill-in in the resulting matrix structure in comparison with the original matrix. The distinctive feature of the ILU0 preconditioner is that it preserves the structure of the original matrix in the result.

Unlike the ILU0 preconditioner, the ILUT preconditioner preserves some resulting fill-in in the preconditioner matrix structure. The distinctive feature of the ILUT algorithm is that it calculates each element of the preconditioner and saves each one if it satisfies two conditions simultaneously: its value is greater than the product of the given tolerance and matrix row norm, and its value is in the given bandwidth of the resulting preconditioner matrix.

Both ILU0 and ILUT preconditioners can apply to any non-degenerate matrix. They can be used alone or together with the Intel® oneAPI Math Kernel Library (oneMKL) RCI FGMRES solver (seeSparse Solver Routines). Avoid using these preconditioners with MKL RCI CG solver because in general, they produce a non-symmetric resulting matrix even if the original matrix is symmetric. Usually, an inverse of the preconditioner is required in this case. To do this the Intel® oneAPI Math Kernel Library (oneMKL) triangular solver routine`mkl_dcsrtrsv` must be applied twice: for the lower triangular part of the preconditioner, and then for its upper triangular part.

NOTE
Although ILU0 and ILUT preconditioners apply to any non-degenerate matrix, in some cases the algorithm may fail to ensure successful termination and the required result. Whether or not the preconditioner produces an acceptable result can only be determined in practice.

A preconditioner may increase the number of iterations for an arbitrary case of the system and the initial solution, and even ruin the convergence. It is your responsibility as a user to choose a suitable preconditioner.

General Scheme of Using ILUT and RCI FGMRES Routines

The general scheme for use is the same for both preconditioners. Some differences exist in the calling parameters of the preconditioners and in the subsequent call of two triangular solvers. You can see all these differences in the preconditioner code examples (`dcsrilu*.*`) in the `examples`folder of the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- examples/solverf/source

The following pseudocode shows the general scheme of using the ILUT preconditioner in the RCI FGMRES context.

```
...
generate matrix A
generate preconditioner C (optional)
  call dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
   change parameters in ipar, dpar if necessary
   call dcsrilut(n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar, ierr)
   call dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
1  call dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
   if (RCI_request.eq.1) then
     multiply the matrix A by tmp(ipar(22)) and put the result in tmp(ipar(23))
c  proceed with FGMRES iterations
    goto 1
   endif
   if (RCI_request.eq.2) then
     do the stopping test
     if (test not passed) then
c proceed with FGMRES iterations
      go to 1
     else
c  stop FGMRES iterations.
      goto 2
     endif
   endif
   if (RCI_request.eq.3) then
c   Below, trvec is an intermediate vector of length at least n
c   Here is the recommended use of the result produced by the ILUT routine.
c   via standard Intel® oneAPI Math Kernel Library (oneMKL) Sparse Blas solver routinemkl_dcsrtrsv.
   call mkl_dcsrtrsv('L','N','U', n, bilut, ibilut, jbilut, tmp(ipar(22)),trvec)
   call mkl_dcsrtrsv('U','N','N', n, bilut, ibilut, jbilut, trvec, tmp(ipar(23)))
c proceed with FGMRES iterations
    goto 1
   endif
   if (RCI_request.eq.4) then
   check the norm of the next orthogonal vector, it is contained in dpar(7)
```

```
    if (the norm is not zero up to rounding/computational errors) then
c  proceed with FGMRES iterations
     goto 1
    else
c  stop FGMRES iterations
     goto 2
    endif
   endif
```

2 call `dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)`

current iteration number is in *itercount*

the computed approximation is in the array *x*

ILU0 and ILUT Preconditioners Interface Description

The concepts required to understand the use of the Intel® oneAPI Math Kernel Library (oneMKL) preconditioner routines are discussed in theAppendix A Linear Solvers Basics.

In this section FORTRAN style notations are used. All types refer to the standard Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

User Data Arrays

The preconditioner routines take arrays of user data as input. To minimize storage requirements and improve overall run-time efficiency, the Intel® oneAPI Math Kernel Library (oneMKL) preconditioner routines do not make copies of the user input arrays.

Common Parameters

Some parameters of the preconditioners are common with the FGMRES Common Parameters. The routine `dfgmres_init` specifies their default and initial values. However, some parameters can be redefined with other values. These parameters are listed below.

For the ILU0 preconditioner:

ipar`(2)` - specifies the destination of error messages generated by the ILU0 routine. The default value 6 means that all error messages are displayed on the screen. Otherwise routine creates a log file called `MKL_PREC_log.txt` and writes error messages to it. Note if the parameter *ipar*`(6)` is set to 0, then error messages are not generated at all.

ipar`(6)` - specifies whether error messages are generated. If its value is not equal to 0, the ILU0 routine returns error messages as specified by the parameter *ipar*`(2)`. Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter *ierr*. The default value is 1.

For the ILUT preconditioner:

ipar`(2)` - specifies the destination of error messages generated by the ILUT routine. The default value 6 means that all messages are displayed on the screen. Otherwise routine creates a log file called `MKL_PREC_log.txt` and writes error messages to it. Note if the parameter *ipar*`(6)` is set to 0, then error messages are not generated at all.

ipar`(6)` - specifies whether error messages are generated. If its value is not equal to 0, the ILUT routine returns error messages as specified by the parameter *ipar*`(2)`. Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter *ierr*. The default value is 1.

ipar(7) - if its value is greater than 0, the ILUT routine generates warning messages as specified by the parameter *ipar*(2) and continues calculations. If its value is equal to 0, the routine returns a positive value of the parameter *ierr*. If its value is less than 0, the routine generates a warning message as specified by the parameter *ipar*(2) and returns a positive value of the parameter *ierr*. The default value is 1.

dcsrilu0

ILU0 preconditioner based on incomplete LU factorization of a sparse matrix.

Syntax

```
call dcsrilu0(n, a, ia, ja, bilu0, ipar, dpar, ierr)
```

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The routine dcsrilu0 computes a preconditioner *B* [Saad03] of a given sparse matrix *A* stored in the format accepted in the direct sparse solvers:

*A~B=L*U* , where *L* is a lower triangular matrix with a unit diagonal, *U* is an upper triangular matrix with a non-unit diagonal, and the portrait of the original matrix *A* is used to store the incomplete factors *L* and *U*.

Caution
This routine supports only one-based indexing of the array parameters.

Input Parameters

n — INTEGER. Size (number of rows or columns) of the original square *n*-by-*n* matrix *A*.

a — DOUBLE PRECISION. Array containing the set of elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to the *values* array description in the Sparse Matrix Storage Format for more details.

ia — INTEGER. Array of size (*n*+1) containing begin indices of rows of the matrix *A* such that *ia*(i) is the index in the array *a* of the first non-zero element from the row i. The value of the last element *ia*(*n*+1) is equal to the number of non-zero elements in the matrix *A*, plus one. Refer to the *rowIndex* array description in the Sparse Matrix Storage Format for more details.

ja — INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. It is important that the indices are in increasing order per row. The matrix size is equal to the size of the array *a*. Refer to the *columns* array description in the Sparse Matrix Storage Format for more details.

Caution
If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

ipar

INTEGER. Array of size *128*. This parameter specifies the integer set of data for both the ILU0 and RCI FGMRES computations. Refer to the *ipar* array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries that are specific to ILU0 are listed below.

ipar(31) specifies how the routine operates when a zero diagonal element occurs during calculation. If this parameter is set to 0 (the default value set by the routine dfgmres_init), then the calculations are stopped and the routine returns a non-zero error value. Otherwise, the diagonal element is set to the value of *dpar*(32) and the calculations continue.

NOTE
You can declare the *ipar* array with a size of 32. However, for future compatibility you must declare the array *ipar* with length 128.

dpar

DOUBLE PRECISION. Array of size *128*. This parameter specifies the double precision set of data for both the ILU0 and RCI FGMRES computations. Refer to the *dpar* array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries specific to ILU0 are listed below.

dpar(31) specifies a small value, which is compared with the computed diagonal elements. When *ipar*(31) is not 0, then diagonal elements less than *dpar*(31) are set to *dpar*(32). The default value is 1.0e-16.

NOTE
This parameter can be set to the negative value, because the calculation uses its absolute value.

If this parameter is set to 0, the comparison with the diagonal element is not performed.

dpar(32) specifies the value that is assigned to the diagonal element if its value is less than *dpar*(31) (see above). The default value is 1.0e-10.

NOTE
You can declare the *dpar* array with a size of 32. However, for future compatibility you must declare the array *dpar* with length 128.

Output Parameters

bilu0 DOUBLE PRECISION. Array *B* containing non-zero elements of the resulting preconditioning matrix *B*, stored in the format accepted in direct sparse solvers. Its size is equal to the number of non-zero elements in the matrix *A*. Refer to the *values* array description in the Sparse Matrix Storage Format section for more details.

ierr INTEGER. Error flag, gives information about the routine completion.

NOTE
To present the resulting preconditioning matrix in the CSR3 format the arrays *ia* (row indices) and *ja* (column indices) of the input matrix must be used.

Return Values

ierr=0 Indicates that the task completed normally.

ierr=-101 Indicates that the routine was interrupted and that error occurred: at least one diagonal element is omitted from the matrix in CSR3 format (see Sparse Matrix Storage Format).

ierr=-102 Indicates that the routine was interrupted because the matrix contains a diagonal element with the value of zero.

ierr=-103 Indicates that the routine was interrupted because the matrix contains a diagonal element which is so small that it could cause an overflow, or that it would cause a bad approximation to ILU0.

ierr=-104 Indicates that the routine was interrupted because the memory is insufficient for the internal work array.

ierr=-105 Indicates that the routine was interrupted because the input matrix size *n* is less than or equal to 0.

ierr=-106 Indicates that the routine was interrupted because the column indices *ja* are not in the ascending order.

Interfaces

FORTRAN 77 and Fortran 95:

```
SUBROUTINE dcsrilu0 (n, a, ia, ja, bilu0, ipar, dpar, ierr)
INTEGER n, ierr, ipar(128)
INTEGER ia(*), ja(*)
DOUBLE PRECISION a(*), bilu0(*), dpar(128)
```

dcsrilut

ILUT preconditioner based on the incomplete LU factorization with a threshold of a sparse matrix.

Syntax

`call dcsrilut(n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar, ierr)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcsrilut` computes a preconditioner *B* [Saad03] of a given sparse matrix *A* stored in the format accepted in the direct sparse solvers:

*A~B=L*U* , where *L* is a lower triangular matrix with unit diagonal and *U* is an upper triangular matrix with non-unit diagonal.

The following threshold criteria are used to generate the incomplete factors *L* and *U*:

1) the resulting entry must be greater than the matrix current row norm multiplied by the parameter *tol*, and

2) the number of the non-zero elements in each row of the resulting *L* and *U* factors must not be greater than the value of the parameter *maxfil*.

> **Caution**
> This routine supports only one-based indexing of the array parameters.

Input Parameters

n

`INTEGER`. Size (number of rows or columns) of the original square *n*-by-*n* matrix *A*.

a

`DOUBLE PRECISION`. Array containing all non-zero elements of the matrix *A*. The length of the array is equal to their number. Refer to *values* array description in the Sparse Matrix Storage Format section for more details.

ia

`INTEGER`. Array of size `(n+1)` containing indices of non-zero elements in the array *a*. *ia*`(i)` is the index of the first non-zero element from the row *i*. The value of the last element *ia*`(n+1)` is equal to the number of non-zeros in the matrix *A*, plus one. Refer to the *rowIndex* array description in the Sparse Matrix Storage Format for more details.

ja

`INTEGER`. Array of size equal to the size of the array *a*. This array contains the column numbers for each non-zero element of the matrix *A*. It is important that the indices are in increasing order per row. Refer to the *columns* array description in the Sparse Matrix Storage Format for more details.

> **Caution**
> If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

tol DOUBLE PRECISION. Tolerance for threshold criterion for the resulting entries of the preconditioner.

maxfil INTEGER. Maximum fill-in, which is half of the preconditioner bandwidth. The number of non-zero elements in the rows of the preconditioner cannot exceed (2*maxfil+1).

ipar INTEGER. Array of size *128*. This parameter is used to specify the integer set of data for both the ILUT and RCI FGMRES computations. Refer to the *ipar* array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries specific to ILUT are listed below.

ipar(31) specifies how the routine operates if the value of the computed diagonal element is less than the current matrix row norm multiplied by the value of the parameter *tol*. If *ipar*(31) = 0, then the calculation is stopped and the routine returns non-zero error value. Otherwise, the value of the diagonal element is set to a value determined by *dpar*(31) (see its description below), and the calculations continue.

NOTE
There is no default value for *ipar*(31) even if the preconditioner is used within the RCI ISS context. Always set the value of this entry.

NOTE
You must declare the array *ipar* with length 128. While defining the array in the code as INTEGER*ipar*(31) works, there is no guarantee of future compatibility with Intel® oneAPI Math Kernel Library (oneMKL).

dpar DOUBLE PRECISION. Array of size *128*. This parameter specifies the double precision set of data for both ILUT and RCI FGMRES computations. Refer to the *dpar* array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries that are specific to ILUT are listed below.

dpar(31) used to adjust the value of small diagonal elements. Diagonal elements with a value less than the current matrix row norm multiplied by *tol* are replaced with the value of *dpar*(31) multiplied by the matrix row norm.

NOTE
There is no default value for *dpar*(31) entry even if the preconditioner is used within RCI ISS context. Always set the value of this entry.

NOTE
You must declare the array *dpar* with length 128. While defining the array in the code as DOUBLE PRECISION*ipar*(31) works, there is no guarantee of future compatibility with Intel® oneAPI Math Kernel Library (oneMKL).

Output Parameters

bilut DOUBLE PRECISION. Array containing non-zero elements of the resulting preconditioning matrix *B*, stored in the format accepted in the direct sparse solvers. Refer to the *values* array description in the Sparse Matrix Storage Format for more details. The size of the array is equal to (2**maxfil*+1)**n*-*maxfil**(*maxfil*+1)+1.

NOTE
Provide enough memory for this array before calling the routine. Otherwise, the routine may fail to complete successfully with a correct result.

ibilut INTEGER. Array of size (*n*+1) containing indices of non-zero elements in the array *bilut*. *ibilut*(i) is the index of the first non-zero element from the row *i*. The value of the last element *ibilut*(*n*+1) is equal to the number of non-zeros in the matrix *B*, plus one. Refer to the *rowIndex* array description in the Sparse Matrix Storage Format for more details.

jbilut INTEGER. Array, its size is equal to the size of the array *bilut*. This array contains the column numbers for each non-zero element of the matrix *B*. Refer to the *columns* array description in the Sparse Matrix Storage Format for more details.

ierr INTEGER. Error flag, gives information about the routine completion.

Return Values

ierr=0 Indicates that the task completed normally.

ierr=-101 Indicates that the routine was interrupted because of an error: the number of elements in some matrix row specified in the sparse format is equal to or less than 0.

ierr=-102	Indicates that the routine was interrupted because the value of the computed diagonal element is less than the product of the given tolerance and the current matrix row norm, and it cannot be replaced as *ipar*(31)=0.
ierr=-103	Indicates that the routine was interrupted because the element *ia*(i + 1) is less than or equal to the element *ia*(i) (see Sparse Matrix Storage Format).
ierr=-104	Indicates that the routine was interrupted because the memory is insufficient for the internal work arrays.
ierr=-105	Indicates that the routine was interrupted because the input value of *maxfil* is less than 0.
ierr=-106	Indicates that the routine was interrupted because the size *n* of the input matrix is less than 0.
ierr=-107	Indicates that the routine was interrupted because an element of the array *ja* is less than 1, or greater than *n* (see Sparse Matrix Storage Format).
ierr=101	The value of *maxfil* is greater than or equal to *n*. The calculation is performed with the value of *maxfil* set to (*n*-1).
ierr=102	The value of *tol* is less than 0. The calculation is performed with the value of the parameter set to (-*tol*)
ierr=103	The absolute value of *tol* is greater than value of *dpar*(31); it can result in instability of the calculation.
ierr=104	The value of *dpar*(31) is equal to 0. It can cause calculations to fail.

Interfaces

FORTRAN 77 and Fortran 95:

```
SUBROUTINE dcsrilut (n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar, ierr)
INTEGER n, ierr, ipar(*), maxfil
INTEGER ia(*), ja(*), ibilut(*), jbilut(*)
DOUBLE PRECISION a(*), bilut(*), dpar(*), tol
```

Sparse Matrix Checker Routines

Intel® oneAPI Math Kernel Library (oneMKL) provides a sparse matrix checker so that you can find errors in the storage of sparse matrices before calling Intel® oneAPI Math Kernel Library (oneMKL) PARDISO, DSS, or Sparse BLAS routines.

sparse_matrix_checker

Checks the correctness of a sparse matrix.

Syntax

error = sparse_matrix_checker (*handle*)

Include Files

- mkl_sparse_handle.fi, mkl_sparse_handle.f90

Description

The sparse_matrix_checker routine checks a user-defined array used to store a sparse matrix in order to detect issues which could cause problems in routines that require sparse input matrices, such as Intel® oneAPI Math Kernel Library (oneMKL) PARDISO, DSS, or Sparse BLAS.

Input Parameters

handle

TYPE (SPARSE_STRUCT), INTENT(INOUT)

Pointer to the data structure describing the sparse array to check.

Return Values

The routine returns a value *error*. Additionally, the *check_result* parameter returns information about where the error occurred, which can be used when *message_level* is MKL_NO_PRINT.

Sparse Matrix Checker Error Values

error value	Meaning	Location
MKL_SPARSE_CHECKER_SUCCESS	The input array successfully passed all checks.	
MKL_SPARSE_CHECKER_NON_MONOTONIC	The input array is not 0 or 1 based (*ia*(1), is not 0 or 1) or elements of *ia* are not in non-decreasing order as required.	**Fortran:** *ia*(*i* + 1) and *ia*(*i* + 2) are incompatible. *check_result*(1) = *i* *check_result*(2) = *ia*(*i* + 1) *check_result*(3) = *ia*(*i* + 2)
MKL_SPARSE_CHECKER_OUT_OF_RANGE	The value of the *ja* array is lower than the number of the first column or greater than the number of the last column.	**Fortran:** *ia*(*i* + 1) and *ia*(*i* + 2) are incompatible. *check_result*(1) = *i* *check_result*(2) = *ia*(*i* + 1) *check_result*(3) = *ia*(*i* + 2)
MKL_SPARSE_CHECKER_NONTRIANGULAR	The *matrix_structure* parameter is MKL_UPPER_TRIANGULAR and both *ia* and *ja* are not upper triangular, or the *matrix_structure* parameter is MKL_LOWER_TRIANGULAR and both *ia* and *ja* are not lower triangular	**Fortran:** *ia*(*i* + 1) and *ja*(*j* + 1) are incompatible. *check_result*(1) = *i* *check_result*(2) = *ia*(*i* + 1) = *j* *check_result*(3) = *ja*(*j* + 1)

error value	Meaning	Location
MKL_SPARSE_CHECKER_NON ORDERED	The elements of the *ja* array are not in non-decreasing order in each row as required.	**Fortran:** *ja*(*j* + 1) and *ja*(*j* + 2) are incompatible. *check_result*(1) = *j* *check_result*(2) = *ja*(*j* + 1) *check_result*(3) = *ja*(*j* + 2)

See Also

sparse_matrix_checker_init Initializes handle for sparse matrix checker.
Intel® oneAPI Math Kernel Library (oneMKL) PARDISO - Parallel Direct Sparse Solver Interface
Sparse BLAS Level 2 and Level 3 Routines
Sparse Matrix Storage Formats

sparse_matrix_checker_init

Initializes handle for sparse matrix checker.

Syntax

```
call sparse_matrix_checker_init (handle)
```

Include Files

- mkl_sparse_handle.fi, mkl_sparse_handle.f90

Description

The sparse_matrix_checker_init routine initializes the handle for the sparse_matrix_checker routine. The *handle* variable contains this data:

Description of sparse_matrix_checker*handle* Data

Field	Type	Possible Values	Meaning
n	INTEGER		Order of the matrix stored in sparse array.
csr_ia	INTEGER (C_INTPTR_T)	Pointer to *ia* array for matrix_format = MKL_CSR	
csr_ja	INTEGER (C_INTPTR_T)	Pointer to *ja* array for matrix_format = MKL_CSR	
check_result(3)	INTEGER (KIND=4)	See Sparse Matrix Checker Error Values for a description of the values returned in *check_result*.	Indicates location of problem in array when message_level = MKL_NO_PRINT.
indexing	INTEGER (KIND=4)	MKL_ZERO_BASED MKL_ONE_BASED	Indexing style used in array.

Field	Type	Possible Values	Meaning
matrix_structure	INTEGER (KIND=4)	MKL_GENERAL_STRUCTURE MKL_UPPER_TRIANGULAR MKL_LOWER_TRIANGULAR MKL_STRUCTURAL_SYMMETRIC	Type of sparse matrix stored in array.
matrix_format	INTEGER (KIND=4)	MKL_CSR	Format of array used for sparse matrix storage.
message_level	INTEGER (KIND=4)	MKL_NO_PRINT MKL_PRINT	Determines whether or not feedback is provided on the screen.
print_style	INTEGER (KIND=4)	MKL_C_STYLE MKL_FORTRAN_STYLE	Determines style of messages when message_level = MKL_PRINT.

Input Parameters

handle TYPE (SPARSE_STRUCT), INTENT(INOUT)

Pointer to the data structure describing the sparse array to check.

Output Parameters

handle Pointer to the initialized data structure.

See Also

sparse_matrix_checker Checks the correctness of a sparse matrix.
Intel® oneAPI Math Kernel Library (oneMKL) PARDISO - Parallel Direct Sparse Solver Interface
Sparse BLAS Level 2 and Level 3 Routines
Sparse Matrix Storage Formats

Extended Eigensolver Routines

NOTE
Intel® oneAPI Math Kernel Library (oneMKL) only supports the shared memory programming (SMP) version of the eigenvalue solver.

- The FEAST Algorithm gives a brief description of the algorithm underlying the Extended Eigensolver.
- Extended Eigensolver Functionality describes the problems that can and cannot be solved with the Extended Eigensolver and how to get the best results from the routines.
- Extended Eigensolver Interfaces gives a reference for calling Extended Eigensolver routines.
-

The FEAST Algorithm

The Extended Eigensolver functionality is a set of high-performance numerical routines for solving symmetric standard eigenvalue problems, $Ax=\lambda x$, or generalized symmetric-definite eigenvalue problems, $Ax=\lambda Bx$. It yields all the eigenvalues (λ) and eigenvectors (x) within a given search interval $[\lambda_{min}, \lambda_{max}]$. It is based on the FEAST algorithm, an innovative fast and stable numerical algorithm presented in [Polizzi09], which fundamentally differs from the traditional Krylov subspace iteration based techniques (Arnoldi and Lanczos algorithms [Bai00]) or other Davidson-Jacobi techniques [Sleijpen96]. The FEAST algorithm is inspired by the density-matrix representation and contour integration techniques in quantum mechanics.

The FEAST numerical algorithm obtains eigenpair solutions using a numerically efficient contour integration technique. The main computational tasks in the FEAST algorithm consist of solving a few independent linear systems along the contour and solving a reduced eigenvalue problem. Consider a circle centered in the middle of the search interval $[\lambda_{min}, \lambda_{max}]$. The numerical integration over the circle in the current version of FEAST is performed using N_e-point Gauss-Legendre quadrature with x_e the e-th Gauss node associated with the weight ω_e. For example, for the case $N_e = 8$:

(x_1, ω_1) = (0.183434642495649 , 0.362683783378361),
(x_2, ω_2) = (-0.183434642495649 , 0.362683783378361),
(x_3, ω_3) = (0.525532409916328 , 0.313706645877887),
(x_4, ω_4) = (-0.525532409916328 , 0.313706645877887),
(x_5, ω_5) = (0.796666477413626 , 0.222381034453374),
(x_6, ω_6) = (-0.796666477413626 , 0.222381034453374),
(x_7, ω_7) = (0.960289856497536 , 0.101228536290376), and
(x_8, ω_8) = (-0.960289856497536 , 0.101228536290376).

The figure FEAST Pseudocode shows the basic pseudocode for the FEAST algorithm for the case of real symmetric (left pane) and complex Hermitian (right pane) generalized eigenvalue problems, using N for the size of the system and M for the number of eigenvalues in the search interval (see [Polizzi09]).

NOTE
The pseudocode presents a simplified version of the actual algorithm. Refer to http://arxiv.org/abs/1302.0432 for an in-depth presentation and mathematical proof of convergence of FEAST.

FEAST Pseudocode

A: real symmetric	A: complex Hermitian
B: symmetric positive definite (SPD)	B: Hermitian positive definite (HPD)
$\Re\{x\}$: real part of x	

1. Select $M_0 > M$ random vectors $Y_{N\times M_0}$
2. Set $Q = 0$ with $Q \in \mathbb{R}^{N\times M_0}$; $r = (\lambda_{max}$ -
   For $e = 1,\ldots,N_e$
   compute $\theta_e = -(\pi / 2)(x_e - 1)$,
   compute $Z_e = (\lambda_{max} + \lambda_{min}) / 2 + r\exp$
   solve $(Z_e B - A)Q_e = Y$ to obtain $Q_e \in$
   compute $Q = Q - (\omega_e / 2)\Re\{r\exp(i\theta_e$
   End
3. Form $A_{Q_{M_0\times M_0}} = Q^T AQ$ and $B_{Q_{M_0\times M_0}} = Q^T$
   reduce value of M_0 if B_Q is not sym
   positive definite.
4. Solve $A_Q\Phi = \varepsilon B_Q\Phi$ to obtain the M_0
   ε_m, and eigenvectors $\Phi_{M_0\times M_0} \in \mathbb{R}^{M_0\times}$
5. Set $\lambda_m = \varepsilon_m$ and compute $X_{N\times M_0} = Q_{N\times}$
   If $\lambda_m \in [\lambda_{min}, \lambda_{max}]$, λ_m is an eigenvalu
   and its eigenvector is X_m (the m-th
6. Check convergence for the trace of
   eigenvalues λ_m. If iterative refinem
   needed, compute $Y = BX$ and go ba

1. Select $M_0 > M$ random vectors $Y_{N\times M_0} \in \mathbb{C}^{N\times M_0}$.
2. Set $Q = 0$ with $Q \in \mathbb{R}^{N\times M_0}$; $r = (\lambda_{max} - \lambda_{min}) / 2$;
   For $e = 1,\ldots,N_e$
   compute $\theta_e = -(\pi / 2)(x_e - 1)$,
   compute $Z_e = (\lambda_{max} + \lambda_{min}) / 2 + r\exp(i\theta_e)$,
   solve $(Z_e B - A)Q_e = Y$ to obtain $Q_e \in \mathbb{C}^{N\times M_0}$
   solve $(Z_e B - A)^H \hat{Q}_e = Y$ to obtain $\hat{Q}_e \in \mathbb{C}^{N\times M_0}$
   $Q = Q - (\omega_e / 4)r\left(\exp(i\theta_e)Q_e + \exp(-i\theta_e)\hat{Q}_e\right)$
   End
3. Form $A_{Q_{M_0\times M_0}} = Q^H AQ$ and $B_{Q_{M_0\times M_0}} = Q^H BQ$
   reduce value of M_0 if B_Q is not Hermitian
   positive definite.
4. Solve $A_Q\Phi = \varepsilon B_Q\Phi$ to obtain the M_0 eigenvalue
   ε_m, and eigenvectors $\Phi_{M_0\times M_0} \in \mathbb{C}^{M_0\times M_0}$.
5. Set $\lambda_m = \varepsilon_m$ and compute $X_{N\times M_0} = Q_{N\times M_0}\Phi_{M_0\times M_0}$.
   If $\lambda_m \in [\lambda_{min}, \lambda_{max}]$, λ_m is an eigenvalue solution
   and its eigenvector is X_m (the m-th column of X).
6. Check convergence for the trace of the
   eigenvalues λ_m. If iterative refinement is
   needed, compute $Y = BX$ and go back to step 2.

Extended Eigensolver Functionality

The eigenvalue problems covered are as follows:

- standard, $Ax = \lambda x$
  - A complex Hermitian
  - A real symmetric
- generalized, $Ax = \lambda Bx$
  - A complex Hermitian, B Hermitian positive definite (hpd)
  - A real symmetric and B real symmetric positive definite (spd)

The Extended Eigensolver functionality offers:

- Real/Complex and Single/Double precisions: double precision is recommended to provide better accuracy of eigenpairs.
- Reverse communication interfaces (RCI) provide maximum flexibility for specific applications. RCI are independent of matrix format and inner system solvers, so you must provide your own linear system solvers (direct or iterative) and matrix-matrix multiply routines.
- Predefined driver interfaces for dense, LAPACK banded, and sparse (CSR) formats are less flexible but are optimized and easy to use:
  - The Extended Eigensolver interfaces for dense matrices are likely to be slower than the comparable LAPACK routines because the FEAST algorithm has a higher computational cost.
  - The Extended Eigensolver interfaces for banded matrices support banded LAPACK-type storage.
  - The Extended Eigensolver sparse interfaces support compressed sparse row format and use the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Parallelism in Extended Eigensolver Routines

How you achieve parallelism in Extended Eigensolver routines depends on which interface you use. Parallelism (via shared memory programming) is not *explicitly* implemented in Extended Eigensolver routines within one node: the inner linear systems are currently solved one after another.

- Using the Extended Eigensolver RCI interfaces, you can achieve parallelism by providing a threaded inner system solver and a matrix-matrix multiplication routine. When using the RCI interfaces, you are responsible for activating the threaded capabilities of your BLAS and LAPACK libraries most likely using the shell variable `OMP_NUM_THREADS`.
- Using the predefined Extended Eigensolver interfaces, parallelism can be implicitly obtained within the shared memory version of BLAS, LAPACK or Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. The shell variable`MKL_NUM_THREADS`can be used for automatically setting the number of OpenMP threads (cores) for BLAS, LAPACK, and Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Achieving Performance With Extended Eigensolver Routines

In order to use the Extended Eigensolver Routines , you need to provide

- the search interval and the size of the subspace M_0 (overestimation of the number of eigenvalues M within a given search interval);
- the system matrix in dense, banded, or sparse CSR format if the Extended Eigensolver predefined interfaces are used, or a high-performance complex direct or iterative system solver and matrix-vector multiplication routine if RCI interfaces are used.

In return, you can expect

- fast convergence with very high accuracy when seeking up to 1000 eigenpairs (in two to four iterations using $M_0 = 1.5M$, and $N_e = 8$ or at most using $N_e = 16$ contour points);
- an extremely robust approach.

The performance of the basic FEAST algorithm depends on a trade-off between the choices of the number of Gauss quadrature points N_e, the size of the subspace M_0, and the number of outer refinement loops to reach the desired accuracy. In practice you should use $M_0 > 1.5\ M$, $N_e = 8$, and at most two refinement loops.

For better performance:

- M_0 should be much smaller than the size of the eigenvalue problem, so that the arithmetic complexity depends mainly on the inner system solver (O(*NM*) for narrow-banded or sparse systems).
- Parallel scalability performance depends on the shared memory capabilities of the of the inner system solver.
- For very large sparse and challenging systems, application users should make use of the Extended Eigensolver RCI interfaces with customized highly-efficient iterative systems solvers and preconditioners.
- For the Extended Eigensolver interfaces for banded matrices, the parallel performance scalability is limited.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Extended Eigensolver Interfaces for Eigenvalues within Interval

Extended Eigensolver Naming Conventions

There are two different types of interfaces available in the Extended Eigensolver routines:

1. The reverse communication interfaces (RCI):

```
?feast_<matrix type>_rci
```

   These interfaces are matrix free format (the interfaces are independent of the matrix data formats). You must provide matrix-vector multiply and direct/iterative linear system solvers for your own explicit or implicit data format.
2. The predefined interfaces:

```
?feast_<matrix type><type of eigenvalue problem>
```

   are predefined drivers for `?feast` reverse communication interface that act on commonly used matrix data storage (dense, banded and compressed sparse row representation), using internal matrix-vector routines and selected inner linear system solvers.

For these interfaces:

- `?` indicates the data type of matrix *A* (and matrix *B* if any) defined as follows:

`s`	real, single precision
`d`	real, double precision
`c`	complex, single precision
`z`	complex , double precision

- *<matrix type>* defined as follows:

Value of *<matrix type>*		**Matrix format**	**Inner linear system solver used by Extended Eigensolver**
`sy`	(symmetric real)	Dense	LAPACK dense solvers
`he`	(Hermitian complex)		
`sb`	(symmetric banded real)	Banded-LAPACK	Internal banded solver

Value of *<matrix type>*		Matrix format	Inner linear system solver used by Extended Eigensolver
hb	(Hermitian banded complex)	Compressed sparse row	PARDISO solver
scsr	(symmetric real)		
hcsr	(Hermitian complex)		
s	(symmetric real)	Reverse communications interfaces	User defined
h	(Hermitian complex)		

- *<type of eigenvalue problem>* is:

gv	generalized eigenvalue problem
ev	standard eigenvalue problem

For example, sfeast_scsrev is a single-precision routine with a symmetric real matrix stored in sparse compressed-row format for a standard eigenvalue problem, and zfeast_hrci is a complex double-precision routine with a Hermitian matrix using the reverse communication interface.

Note that:

- ? can be s or d if a matrix is real symmetric: *<matrix type>* is sy, sb, or scsr.
- ? can be c or z if a matrix is complex Hermitian: *<matrix type>* is he, hb, or hcsr.
- ? can be c or z if the Extended Eigensolver RCI interface is used for solving a complex Hermitian problem.
- ? can be s or d if the Extended Eigensolver RCI interface is used for solving a real symmetric problem.

feastinit

Initialize Extended Eigensolver input parameters with default values.

Syntax

```
call feastinit (fpm)
```

Include Files

- mkl.fi

Description

This routine sets all Extended Eigensolver parameters to their default values.

Output Parameters

fpm	INTEGER Array, size 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

Extended Eigensolver Input Parameters

The input parameters for Extended Eigensolver routines are contained in an integer array named *fpm*. To call the Extended Eigensolver interfaces, this array should be initialized using the routine feastinit.

Parameter	Default	Description	
fpm(1)	0	Specifies whether Extended Eigensolver routines print runtime status.	
		fpm(1)=0	Extended Eigensolver routines do not generate runtime messages at all.
		fpm(1)=1	Extended Eigensolver routines print runtime status to the screen.
fpm(2)	8	The number of contour points N_e = 8 (see the description of FEAST algorithm). Must be one of {3,4,5,6,8,10,12,16,20,24,32,40,48}.	
fpm(3)	12	Error trace double precision stopping criteria ε ($\varepsilon = 10^{-fpm(3)}$) .	
fpm(4)	20	Maximum number of Extended Eigensolver refinement loops allowed. If no convergence is reached within *fpm*(4) refinement loops, Extended Eigensolver routines return *info*=2.	
fpm(5)	0	User initial subspace. If *fpm*(5)=0 then Extended Eigensolver routines generate initial subspace, if *fpm*(5)=1 the user supplied initial subspace is used.	
fpm(6)	0	Extended Eigensolver stopping test.	
		fpm(6)=0	Extended Eigensolvers are stopped if this residual stopping test is satisfied: • generalized eigenvalue problem: $\max_{i=1:mode} \frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(\|E_{min}\|, \|E_{max}\|)\|Bx_i\|_1} < \varepsilon$ • standard eigenvalue problem: $\max_{i=1:mode} \frac{\|Ax_i - \lambda_i x_i\|_1}{\max(\|E_{min}\|, \|E_{max}\|)\|x_i\|_1} < \varepsilon$ where *mode* is the total number of eigenvalues found in the search interval and $\varepsilon = 10^{-fpm(7)}$ for real and complex or $\varepsilon = 10^{-fpm(3)}$ for double precision and double complex.
		fpm(6)=1	Extended Eigensolvers are stopped if this trace stopping test is satisfied: $\frac{\|trace_j - trace_{j-1}\|}{\max(\|E_{min}\|, \|E_{max}\|)} < \varepsilon$, where $trace_j$ denotes the sum of all eigenvalues found in the search interval [*emin*, *emax*] at the *j*-th Extended Eigensolver iteration: $trace_j = \sum_{i=1}^{M} \lambda_i^{(j)}$.
fpm(7)	5	Error trace single precision stopping criteria ($10^{-fpm(7)}$) .	
fpm(14)	0	*fpm*(14)=0	Standard use for Extended Eigensolver routines.

Parameter	Default	Description	
		fpm(14)=1	Non-standard use for Extended Eigensolver routines: return the computed eigenvectors subspace after one single contour integration.
fpm(27)	0	Specifies whether Extended Eigensolver routines check input matrices (applies to CSR format only).	
		fpm(27)=0	Extended Eigensolver routines do not check input matrices.
		fpm(27)=1	Extended Eigensolver routines check input matrices.
fpm(28)	0	Check if matrix *B* is positive definite. Set *fpm*(28) = 1 to check if *B* is positive definite.	
fpm(30) to *fpm*(63)	-	Reserved for future use.	
fpm(64)	0	Use the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver with the user-defined PARDISO *iparm* array settings.	
		NOTE This option can only be used by Extended Eigensolver Predefined Interfaces for Sparse Matrices.	
		fpm(64)=0	Extended Eigensolver routines use the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO default *iparm* settings defined by calling the pardisoinit subroutine.
		fpm(64)=1	The values from *fpm*(65) to *fpm*(128) correspond to *iparm*(1) to *iparm*(64) respectively according to the formula *fpm*(64 + *i*) = *iparm*(*i*) for *i* = 1, 2, ..., 64.

Extended Eigensolver Output Details

Errors and warnings encountered during a run of the Extended Eigensolver routines are stored in an integer variable, *info*. If the value of the output *info* parameter is not 0, either an error or warning was encountered. The possible return values for the *info* parameter along with the error code descriptions are given in the following table.

Return Codes for info Parameter

info	Classification	Description
202	Error	Problem with size of the system *n* (*n*≤0)
201	Error	Problem with size of initial subspace *m0* (*m0*≤0 or *m0*>*n*)
200	Error	Problem with *emin*,*emax* (*emin*≥*emax*)
(100+*i*)	Error	Problem with *i*-th value of the input Extended Eigensolver parameter (*fpm*(*i*)). Only the parameters in use are checked.
4	Warning	Successful return of only the computed subspace after call with *fpm*(14) = 1

info	Classification	Description
3	Warning	Size of the subspace *m0* is too small (*m0*<*m*)
2	Warning	No Convergence (number of iteration loops > *fpm*(4))
1	Warning	No eigenvalue found in the search interval. See remark below for further details.
0	Successful exit	
-1	Error	Internal error for allocation memory.
-2	Error	Internal error of the inner system solver. Possible reasons: not enough memory for inner linear system solver or inconsistent input.
-3	Error	Internal error of the reduced eigenvalue solver Possible cause: matrix *B* may not be positive definite. It can be checked by setting *fpm*(28) = 1 before calling an Extended Eigensolver routine, or by using LAPACK routines.
-4	Error	Matrix *B* is not positive definite.
-(100+*i*)	Error	Problem with the *i*-th argument of the Extended Eigensolver interface.

In some extreme cases the return value *info*=1 may indicate that the Extended Eigensolver routine has failed to find the eigenvalues in the search interval. This situation could arise if a very large search interval is used to locate a small and isolated cluster of eigenvalues (i.e. the dimension of the search interval is many orders of magnitude larger than the number of contour points. It is then either recommended to increase the number of contour points *fpm*(2) or simply rescale more appropriately the search interval. Rescaling means the initial problem of finding all eigenvalues the search interval $[\lambda_{min},\lambda_{max}]$ for the standard eigenvalue problem $A\,x=\lambda x$ is replaced with the problem of finding all eigenvalues in the search interval $[\lambda_{min}/\,t,\,\lambda_{max}/\,t]$ for the standard eigenvalue problem $(A/t)\,x=(\lambda/t)\,x$ where t is a scaling factor.

Extended Eigensolver RCI Routines

If you do not require specific linear system solvers or matrix storage schemes, you can skip this section and go directly to Extended Eigensolver Predefined Interfaces.

Extended Eigensolver RCI Interface Description

The Extended Eigensolver RCI interfaces can be used to solve standard or generalized eigenvalue problems, and are independent of the format of the matrices. As mentioned earlier, the Extended Eigensolver algorithm is based on the contour integration techniques of the matrix resolvent $G(\sigma)= (\sigma B - A)^{-1}$ over a circle. For solving a generalized eigenvalue problem, Extended Eigensolver has to perform one or more of the following operations at each contour point denoted below by Z_e :

- Factorize the matrix $(Z_e * B - A)$
- Solve the linear system $(Z_e * B - A)X = Y$ or $(Z_e * B - A)^{H}X = Y$ with multiple right hand sides, where H means transpose conjugate
- Matrix-matrix multiply $BX = Y$ or $AX = Y$

For solving a standard eigenvalue problem, replace the matrix B with the identity matrix I.

The primary aim of RCI interfaces is to isolate these operations: the linear system solver, factorization of the matrix resolvent at each contour point, and matrix-matrix multiplication. This gives universality to RCI interfaces as they are independent of data structures and the specific implementation of the operations like matrix-vector multiplication or inner system solvers. However, this approach requires some additional effort when calling the interface. In particular, operations listed above are performed by routines that you supply on data structures that you find most appropriate for the problem at hand.

To initialize an Extended Eigensolver RCI routine, set the job indicator (*ijob*) parameter to the value -1. When the routine requires the results of an operation, it generates a special value of *ijob* to indicate the operation that needs to be performed. The routine also returns *ze*, the coordinate along the complex contour, the values of array *work* or *workc*, and the number of columns to be used. Your subroutine then must perform the operation at the given contour point *ze*, store the results in prescribed array, and return control to the Extended Eigensolver RCI routine.

The following pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a real symmetric problem:

```
     Ijob=-1 ! initialization
    do while (ijob/=0)
       call  ?feast_srci(ijob, N, Ze, work1, work2, Aq,  Bq,
                   &fpm, epsout, loop, Emin, Emax, M0, E, lambda, q, res, info)

       select case(ijob)
       case(10) !! Factorize the complex matrix (ZeB-A)
 . . . . . . . . . . . . . . . . <<< user entry

       case(11) !! Solve the complex linear system (ZeB-A)x=work2(1:N,1:M0) result in work2
. . . . . . . . . . . . . . . . <<< user entry

       case(30) !! Perform multiplication A*q(1:N,i:j) result in work1(1:N,i:j)
                !!  where i=fpm(24) and j=fpm(24)+fpm(25)-1
. . . . . . . . . . . . . . . . <<< user entry
       case(40) !! Perform multiplication B*q(1:N,i:j) result in work1(1:N,i:j)
                !!  where i=fpm(24) and j=fpm(24)+fpm(25)-1
. . . . . . . . . . . . . . . . <<< user entry

    end select
end do
```

NOTE
The `?` option in `?feast` in the pseudocode given above should be replaced by either `s` or `d`, depending on the matrix data type of the eigenvalue system.

The next pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a complex Hermitian problem:

```
    Ijob=-1 ! initialization
    do while (ijob/=0)
       call  ?feast_hrci(ijob, N, Ze, work1, work2, Aq,  Bq,
                   &fpm, epsout, loop, Emin, Emax, M0, E, lambda, q, res, info)

       select case(ijob)
       case(10) !! Factorize the complex matrix (ZeB-A)
 . . . . . . . . . . . . . . . . <<< user entry
       case (11)!! Solve the linear system (ZeB-A)y=work2 (1:N, 1:M0) result in work2
. . . . . . . . . . . . . . . . <<< user entry
       case (20)!! Factorize ( if needed by case (21)) the complex matrix     (ZeB-A)^H
    !!ATTENTION: This option requires additional memory storage
    !! (i.e . the resulting matrix from case (10) cannot be overwritten)
. . . . . . . . . . . . . . . . <<< user entry
       case (21) !! Solve the linear system (ZeB-A)^Hy=work2(1:N, 1:M0) result in  work2
    !!REMARK: case (20) becomes obsolete if this solve can be performed
    !! using the factorization in case (10)
. . . . . . . . . . . . . . . . <<< user entry
       case(30) !! Perform multiplication A*q(1:N,i:j) result in work1(1:N,i:j)
```

```
                !!  where i=fpm(24) and j=fpm(25)+fpm(24)-1
. . . . . . . . . . . . . . . . . <<< user entry
       case(40) !! Perform multiplication B*q(1:N,i:j) result in work1(1:N,i:j)
                !!  where i=fpm(24) and j=fpm(25)+fpm(24)-1
. . . . . . . . . . . . . . . . . <<< user entry

    end select
end do
```

NOTE
The `?` option in `?feast` in the pseudocode given above should be replaced by either `c` or `z`, depending on the matrix data type of the eigenvalue system.

If `case(20)` can be avoided, performance could be up to twice as fast, and Extended Eigensolver functionality would use half of the memory.

If an iterative solver is used along with a preconditioner, the factorization of the preconditioner could be performed with *ijob* = 10 (and *ijob* = 20 if applicable) for a given value of Z_e, and the associated iterative solve would then be performed with *ijob* = 11 (and *ijob* = 21 if applicable).

?feast_srci/?feast_hrci

Extended Eigensolver RCI interface.

Syntax

```
call sfeast_srci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,
lambda, q, m, res, info)

call dfeast_srci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,
lambda, q, m, res, info)

call cfeast_hrci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,
lambda, q, m, res, info)

call zfeast_hrci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,
lambda, q, m, res, info)
```

Include Files

- `mkl.fi`

Description

Compute eigenvalues as described in Extended Eigensolver RCI Interface Description.

Input Parameters

ijob — INTEGER

Job indicator variable. On entry, a call to `?feast_srci`/`?feast_hrci` with *ijob*=-1 initializes the eigensolver.

n

INTEGER

Sets the size of the problem. *n* > 0.

work

REAL for sfeast_srci

DOUBLE PRECISION for dfeast_srci

COMPLEX for cfeast_hrci

COMPLEX*16 for zfeast_hrci

Workspace array of size *n* by *m0*.

workc

COMPLEX for sfeast_srci and cfeast_hrci

COMPLEX*16 for dfeast_srci and zfeast_hrci

Workspace array of size *n* by *m0*.

aq, *sq*

REAL for sfeast_srci

DOUBLE PRECISION for dfeast_srci

COMPLEX for cfeast_hrci

COMPLEX*16 for zfeast_hrci

Workspace arrays of size *m0* by *m0*.

fpm

INTEGER

Array, size of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

emin, *emax*

REAL for sfeast_srci and cfeast_hrci

DOUBLE PRECISION for dfeast_srci and zfeast_hrci

The lower and upper bounds of the interval to be searched for eigenvalues; *emin* ≤ *emax*.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0

INTEGER

On entry, specifies the initial guess for subspace size to be used, 0 < *m0*≤*n*. Set *m0* ≥ *m* where *m* is the total number of eigenvalues located in the interval [*emin*, *emax*]. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

q

REAL for sfeast_srci

DOUBLE PRECISION for dfeast_srci

COMPLEX for cfeast_hrci

COMPLEX*16 for zfeast_hrci

On entry, if *fpm*(5)=1, the array *q* of size *n* by *m* contains a basis of guess subspace where *n* is the order of the input matrix.

Output Parameters

ijob

On exit, the parameter carries the status flag that indicates the condition of the return. The status information is divided into three categories:

1. A zero value indicates successful completion of the task.
2. A positive value indicates that the solver requires a matrix-vector multiplication or solving a specific system with a complex coefficient.
3. A negative value indicates successful initiation.

A non-zero value of *ijob* specifically means the following:

- *ijob* = 10 - factorize the complex matrix Z_e*B - A at a given contour point Z_e and return the control to the ?feast_srci/?feast_hrci routine where Z_e is a complex number meaning contour point and its value is defined internally in ?feast_srci/?feast_hrci.
- *ijob* =11 - solve the complex linear system (Z_e*B - A)*y = *workc*(*n*, *m0*), put the solution in *workc*(*n*, *m0*) and return the control to the ?feast_srci/?feast_hrci routine.
- *ijob* =20 - factorize the complex matrix (Z_e*B - A)^H at a given contour point Z_e and return the control to the ?feast_srci/?feast_hrci routine where Z_e is a complex number meaning contour point and its value is defined internally in ?feast_srci/?feast_hrci.

  The symbol X^H means transpose conjugate of matrix *X*.
- *ijob* = 21 - solve the complex linear system(Z_e*B - A)^H*y = *workc*(*n*, *m0*), put the solution in *workc*(*n*, *m0*) and return the control to the ?feast_srci/?feast_hrci routine. The case *ijob*=20 becomes obsolete if the solve can be performed using the factorization computed for *ijob*=10.

  The symbol X^H mean transpose conjugate of matrix *X*.
- *ijob* = 30 - multiply matrix *A* by q(*n*, *i*:*j*), put the result in *work*(*n*, *i*:*j*), and return the control to the ?feast_srci/?feast_hrci routine.

  i is *fpm*(25), and *j* is *fpm*(24) + *fpm*(25) - 1.
- *ijob* = 40 - multiply matrix *B* by q(*n*, *i*:*j*), put the result in *work*(*n*, *i*:*j*) and return the control to the ?feast_srci/?feast_hrci routine. If a standard eigenvalue problem is solved, just return *work* = q.

  i is *fpm*(25), and *j* is *fpm*(24) + *fpm*(25) - 1.
- *ijob* = -2 - rerun the ?feast_srci/?feast_hrci task with the same parameters.

ze

COMPLEX for sfeast_srci and cfeast_hrci

COMPLEX*16 for dfeast_srci and zfeast_hrci

Defines the coordinate along the complex contour. All values of *ze* are generated by ?feast_srci/?feast_hrci internally.

fpm

On output, contains coordinates of columns of work array needed for iterative refinement. (See Extended Eigensolver RCI Interface Description.)

epsout
: REAL for sfeast_srci and cfeast_hrci

DOUBLE PRECISION for dfeast_srci and zfeast_hrci

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

loop
: INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

lambda
: REAL for sfeast_srci and cfeast_hrci

DOUBLE PRECISION for dfeast_srci and zfeast_hrci

Array of length *m0*. On output, the first *m* entries of *lambda* are eigenvalues found in the interval.

q
: On output, *q* contains all eigenvectors corresponding to *lambda*.

m
: INTEGER

The total number of eigenvalues found in the interval [*emin*, *emax*]: $0 \le m \le m0$.

res
: REAL for sfeast_srci and cfeast_hrci

DOUBLE PRECISION for dfeast_srci and zfeast_hrci

Array of length *m0*. On exit, the first *m* components contain the relative residual vector:

- generalized eigenvalue problem:

$$\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(|E_{\min}|, |E_{\max}|)\|Bx_i\|_1}$$

- standard eigenvalue problem:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max(|E_{\min}|, |E_{\max}|)\|x_i\|_1}$$

for $i=1, 2, \ldots, m$, and where *m* is the total number of eigenvalues found in the search interval.

info
: INTEGER

If *info*=0, the execution is successful. If *info* ≠ 0, see Output Eigensolver info Details.

Extended Eigensolver Predefined Interfaces

The predefined interfaces include routines for standard and generalized eigenvalue problems, and for dense, banded, and sparse matrices.

Matrix Type	Standard Eigenvalue Problem	Generalized Eigenvalue Problem
Dense	?feast_syev	?feast_sygv

Matrix Type	Standard Eigenvalue Problem	Generalized Eigenvalue Problem
	?feast_heev	?feast_hegv
Banded	?feast_sbev ?feast_hbev	?feast_sbgv ?feast_hbgv
Sparse	?feast_scsrev ?feast_hcsrev	?feast_scsrgv ?feast_hcsrgv

Matrix Storage

The symmetric and Hermitian matrices used in Extended Eigensolvers predefined interfaces can be stored in full, band, and sparse formats.

- In the full storage format (described in Full Storage in additional detail) you store all elements, all of the elements in the upper triangle of the matrix, or all of the elements in the lower triangle of the matrix.
- In the band storage format (described in Band storage in additional detail), you store only the elements along a diagonal band of the matrix.
- In the sparse format (described in Storage Arrays for a Matrix in CSR Format (3-Array Variation)), you store only the non-zero elements of the matrix.

In generalized eigenvalue systems you must use the same family of storage format for both matrices *A* and *B*. The bandwidth can be different for the banded format (*klb* can be different from *kla*), and the position of the non-zero elements can also be different for the sparse format (CSR coordinates *ib* and *jb* can be different from *ia* and *ja*).

?feast_syev/?feast_heev

Extended Eigensolver interface for standard eigenvalue problem with dense matrices.

Syntax

```
call sfeast_syev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call dfeast_syev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call cfeast_heev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call zfeast_heev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

Include Files

- mkl.fi

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

uplo CHARACTER*1

Must be 'U' or 'L' or 'F' .

If *uplo* = 'U', *a* stores the upper triangular parts of *A*.

If *uplo* = 'L', *a* stores the lower triangular parts of *A*.

If *uplo*= 'F' , *a* stores the full matrix *A*.

n

INTEGER

Sets the size of the problem. *n* > 0.

a

REAL for sfeast_syev

DOUBLE PRECISION for dfeast_syev

COMPLEX for cfeast_heev

COMPLEX*16 for zfeast_heev

Array of dimension *lda* by *n*, contains either full matrix *A* or upper or lower triangular part of the matrix *A*, as specified by *uplo*

lda

INTEGER

The leading dimension of the array *a*. Must be at least max(1, *n*).

fpm

INTEGER

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

emin, *emax*

REAL for sfeast_syev and cfeast_heev

DOUBLE PRECISION for dfeast_syev and zfeast_heev

The lower and upper bounds of the interval to be searched for eigenvalues; *emin* ≤ *emax*.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0

INTEGER

On entry, specifies the initial guess for subspace dimension to be used, 0 < *m0*≤*n*. Set *m0* ≥ *m* where *m* is the total number of eigenvalues located in the interval [*emin*, *emax*]. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

x

REAL for sfeast_syev

DOUBLE PRECISION for dfeast_syev

COMPLEX for cfeast_heev

COMPLEX*16 for zfeast_heev

On entry, if *fpm*(5) =1, the array *x*(*n*, *m*) contains a basis of guess subspace where *n* is the order of the input matrix.

Output Parameters

epsout

REAL for sfeast_syev and cfeast_heev

DOUBLE PRECISION for dfeast_syev and zfeast_heev

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

loop

INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

e

REAL for sfeast_syev and cfeast_heev

DOUBLE PRECISION for dfeast_syev and zfeast_heev

Array of length *m0*. On output, the first *m* entries of *e* are eigenvalues found in the interval.

x

On output, the first *m* columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the *i*-th column of *x* holding the eigenvector associated with *e*(*i*).

m

INTEGER

The total number of eigenvalues found in the interval [*emin*, *emax*]: $0 \le m \le m0$.

res

REAL for sfeast_syev and cfeast_heev

DOUBLE PRECISION for dfeast_syev and zfeast_heev

Array of length *m0*. On exit, the first *m* components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max\left(|E_{\min}|, |E_{\max}|\right)\|x_i\|_1}$$

for $i=1, 2, \ldots, m$, and where *m* is the total number of eigenvalues found in the search interval.

info

INTEGER

If *info*=0, the execution is successful. If *info* ≠ 0, see Output Eigensolver info Details.

?feast_sygv/?feast_hegv

Extended Eigensolver interface for generalized eigenvalue problem with dense matrices.

Syntax

```
call sfeast_sygv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call dfeast_sygv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call cfeast_hegv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
```

```
call zfeast_hegv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
```

Include Files

- mkl.fi

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

Input Parameters

uplo

CHARACTER*1

Must be 'U' or 'L' or 'F' .

If *UPLO* = 'U', *a* and *b* store the upper triangular parts of *A* and *B* respectively.

If *UPLO* = 'L', *a* and *b* store the lower triangular parts of *A* and *B* respectively.

If *UPLO*= 'F', *a* and *b* store the full matrices *A* and *B* respectively.

n

INTEGER

Sets the size of the problem. *n* > 0.

a

REAL for sfeast_sygv

DOUBLE PRECISION for dfeast_sygv

COMPLEX for cfeast_hegv

COMPLEX*16 for zfeast_hegv

Array of dimension *lda* by *n*, contains either full matrix *A* or upper or lower triangular part of the matrix *A*, as specified by *uplo*

lda

INTEGER

The leading dimension of the array *a*. Must be at least max(1, *n*).

b

REAL for sfeast_sygv

DOUBLE PRECISION for dfeast_sygv

COMPLEX for cfeast_hegv

COMPLEX*16 for zfeast_hegv

Array of dimension *ldb* by *n*, contains either full matrix *B* or upper or lower triangular part of the matrix *B*, as specified by *uplo*

ldb

INTEGER

The leading dimension of the array *B*. Must be at least max(1, *n*).

fpm

INTEGER

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

emin, *emax*

REAL for sfeast_sygv and cfeast_hegv

DOUBLE PRECISION for dfeast_sygv and zfeast_hegv

The lower and upper bounds of the interval to be searched for eigenvalues; *emin* ≤ *emax*.

> **NOTE** Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0

INTEGER

On entry, specifies the initial guess for subspace dimension to be used, 0 < *m0*≤*n*. Set *m0* ≥ *m* where *m* is the total number of eigenvalues located in the interval [*emin*, *emax*]. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

x

REAL for sfeast_sygv

DOUBLE PRECISION for dfeast_sygv

COMPLEX for cfeast_hegv

COMPLEX*16 for zfeast_hegv

On entry, if *fpm*(5)=1, the array *x*(*n*, *m*) contains a basis of guess subspace where *n* is the order of the input matrix.

Output Parameters

epsout

REAL for sfeast_sygv and cfeast_hegv

DOUBLE PRECISION for dfeast_sygv and zfeast_hegv

On output, contains the relative error on the trace: |$trace_i$ - $trace_{i-1}$| /max(|*emin*|, |*emax*|)

loop

INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

e

REAL for sfeast_sygv and cfeast_hegv

DOUBLE PRECISION for dfeast_sygv and zfeast_hegv

Array of length *m0*. On output, the first *m* entries of *e* are eigenvalues found in the interval.

x

On output, the first *m* columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the *i*-th column of *x* holding the eigenvector associated with *e*(*i*).

m INTEGER

The total number of eigenvalues found in the interval [*emin*, *emax*]: 0 ≤ *m* ≤ *m0*.

res REAL for sfeast_sygv and cfeast_hegv

DOUBLE PRECISION for dfeast_sygv and zfeast_hegv

Array of length *m0*. On exit, the first *m* components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max\left(|E_{\min}|, |E_{\max}|\right)\|Bx_i\|_1}$$

for *i*=1, 2, …, *m*, and where *m* is the total number of eigenvalues found in the search interval.

info INTEGER

If *info*=0, the execution is successful. If *info* ≠ 0, see Output Eigensolver info Details.

?feast_sbev/?feast_hbev

Extended Eigensolver interface for standard eigenvalue problem with banded matrices.

Syntax

```
call sfeast_sbev(uplo, n, kla, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call dfeast_sbev(uplo, n, kla, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call cfeast_hbev(uplo, n, kla, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call zfeast_hbev(uplo, n, kla, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

Include Files

- mkl.fi

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

uplo CHARACTER*1

Must be 'U' or 'L' or 'F' .

If *uplo* = 'U', *a* stores the upper triangular parts of *A*.

If *uplo* = 'L', *a* stores the lower triangular parts of *A*.

If *uplo*= 'F' , *a* stores the full matrix *A*.

n

INTEGER

Sets the size of the problem. *n* > 0.

kla

INTEGER

The number of super- or sub-diagonals within the band in *A* (*kla*≥ 0).

a

REAL for sfeast_sbev

DOUBLE PRECISION for dfeast_sbev

COMPLEX for cfeast_hbev

COMPLEX*16 for zfeast_hbev

Array of dimension *lda* by *n*, contains either full matrix *A* or upper or lower triangular part of the matrix *A*, as specified by *uplo*

lda

INTEGER

The leading dimension of the array *a*. Must be at least max(1, *n*).

fpm

INTEGER

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

emin, *emax*

REAL for sfeast_sbev and cfeast_hbev

DOUBLE PRECISION for dfeast_sbev and zfeast_hbev

The lower and upper bounds of the interval to be searched for eigenvalues; *emin* ≤ *emax*.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0

INTEGER

On entry, specifies the initial guess for subspace dimension to be used, 0 < *m0*≤*n*. Set *m0* ≥ *m* where *m* is the total number of eigenvalues located in the interval [*emin*, *emax*]. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

x

REAL for sfeast_sbev

DOUBLE PRECISION for dfeast_sbev

COMPLEX for cfeast_hbev

COMPLEX*16 for zfeast_hbev

On entry, if *fpm*(5)=1, the array *x*(*n*, *m*) contains a basis of guess subspace where *n* is the order of the input matrix.

Output Parameters

epsout

REAL for sfeast_sbev and cfeast_hbev

DOUBLE PRECISION for dfeast_sbev and zfeast_hbev

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

loop

INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

e

REAL for sfeast_sbev and cfeast_hbev

DOUBLE PRECISION for dfeast_sbev and zfeast_hbev

Array of length *m0*. On output, the first *m* entries of *e* are eigenvalues found in the interval.

x

On output, the first *m* columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the *i*-th column of *x* holding the eigenvector associated with *e*(*i*).

m

INTEGER

The total number of eigenvalues found in the interval [*emin*, *emax*]: $0 \le m \le m0$.

res

REAL for sfeast_sbev and cfeast_hbev

DOUBLE PRECISION for dfeast_sbev and zfeast_hbev

Array of length *m0*. On exit, the first *m* components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max\left(|E_{\min}|, |E_{\max}|\right)\|x_i\|_1}$$

for *i*=1, 2, …, *m*, and where *m* is the total number of eigenvalues found in the search interval.

info

INTEGER

If *info*=0, the execution is successful. If *info* ≠ 0, see Output Eigensolver info Details.

?feast_sbgv/?feast_hbgv

Extended Eigensolver interface for generalized eigenvalue problem with banded matrices.

Syntax

```
call sfeast_sbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call dfeast_sbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call cfeast_hbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call zfeast_hbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

Include Files

- mkl.fi

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

> **NOTE**
> Both matrices *A* and *B* must use the same family of storage format. The bandwidth, however, can be different (*klb* can be different from *kla*).

Input Parameters

uplo

CHARACTER*1

Must be 'U' or 'L' or 'F' .

If *UPLO* = 'U', *a* and *b* store the upper triangular parts of *A* and *B* respectively.

If *UPLO* = 'L', *a* and *b* store the lower triangular parts of *A* and *B* respectively.

If *UPLO*= 'F', *a* and *b* store the full matrices *A* and *B* respectively.

n

INTEGER

Sets the size of the problem. *n* > 0.

kla

INTEGER

The number of super- or sub-diagonals within the band in *A* (*kla*≥ 0).

a

REAL for sfeast_sbgv

DOUBLE PRECISION for dfeast_sbgv

COMPLEX for cfeast_hbgv

COMPLEX*16 for zfeast_hbgv

Array of dimension *lda* by *n*, contains either full matrix *A* or upper or lower triangular part of the matrix *A*, as specified by *uplo*

lda

INTEGER

The leading dimension of the array *a*. Must be at least max(1, *n*).

klb

INTEGER

The number of super- or sub-diagonals within the band in *B* (*klb*≥ 0).

b

REAL for sfeast_sbgv

DOUBLE PRECISION for dfeast_sbgv

COMPLEX for cfeast_hbgv

COMPLEX*16 for zfeast_hbgv

Array of dimension *ldb* by *n*, contains either full matrix *B* or upper or lower triangular part of the matrix *B*, as specified by *uplo*

ldb

INTEGER

The leading dimension of the array *B*. Must be at least max(1, *n*).

fpm

INTEGER

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

emin, *emax*

REAL for sfeast_sbgv and cfeast_hbgv

DOUBLE PRECISION for dfeast_sbgv and zfeast_hbgv

The lower and upper bounds of the interval to be searched for eigenvalues; *emin* ≤ *emax*.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0

INTEGER

On entry, specifies the initial guess for subspace dimension to be used, 0 < *m0*≤*n*. Set *m0* ≥ *m* where *m* is the total number of eigenvalues located in the interval [*emin*, *emax*]. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

x

REAL for sfeast_sbgv

DOUBLE PRECISION for dfeast_sbgv

COMPLEX for cfeast_hbgv

COMPLEX*16 for zfeast_hbgv

On entry, if *fpm*(5)=1, the array *x*(*n*, *m*) contains a basis of guess subspace where *n* is the order of the input matrix.

Output Parameters

epsout

REAL for sfeast_sbgv and cfeast_hbgv

DOUBLE PRECISION for dfeast_sbgv and zfeast_hbgv

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

loop INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

e REAL for sfeast_sbgv and cfeast_hbgv

DOUBLE PRECISION for dfeast_sbgv and zfeast_hbgv

Array of length *m0*. On output, the first *m* entries of *e* are eigenvalues found in the interval.

x On output, the first *m* columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the *i*-th column of *x* holding the eigenvector associated with *e*(*i*).

m INTEGER

The total number of eigenvalues found in the interval [*emin*, *emax*]: 0 ≤ *m* ≤ *m0*.

res REAL for sfeast_sbgv and cfeast_hbgv

DOUBLE PRECISION for dfeast_sbgv and zfeast_hbgv

Array of length *m0*. On exit, the first *m* components contain the relative residual vector:

$$\frac{\left\|Ax_i - \lambda_i Bx_i\right\|_1}{\max\left(\left|E_{\min}\right|, \left|E_{\max}\right|\right)\left\|Bx_i\right\|_1}$$

for *i*=1, 2, …, *m*, and where *m* is the total number of eigenvalues found in the search interval.

info INTEGER

If *info*=0, the execution is successful. If *info* ≠ 0, see Output Eigensolver info Details.

?feast_scsrev/?feast_hcsrev

Extended Eigensolver interface for standard eigenvalue problem with sparse matrices.

Syntax

```
call sfeast_scsrev(uplo, n, a, ia, ja, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call dfeast_scsrev(uplo, n, a, ia, ja, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call cfeast_hcsrev(uplo, n, a, ia, ja, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call zfeast_hcsrev(uplo, n, a, ia, ja, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

Include Files

- mkl.fi

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

uplo

CHARACTER*1

Must be 'U' or 'L' or 'F' .

If *uplo* = 'U', *a* stores the upper triangular parts of *A*.

If *uplo* = 'L', *a* stores the lower triangular parts of *A*.

If *uplo*= 'F' , *a* stores the full matrix *A*.

n

INTEGER

Sets the size of the problem. *n* > 0.

a

REAL for sfeast_scsrev

DOUBLE PRECISION for dfeast_scsrev

COMPLEX for cfeast_hcsrev

COMPLEX*16 for zfeast_hcsrev

Array containing the nonzero elements of either the full matrix *A* or the upper or lower triangular part of the matrix *A*, as specified by *uplo*.

ia

INTEGER

Array of length *n* + 1, containing indices of elements in the array *a* , such that *ia(i)* is the index in the array *a* of the first non-zero element from the row *i* . The value of the last element *ia(n + 1)* is equal to the number of non-zeros plus one.

ja

INTEGER

Array containing the column indices for each non-zero element of the matrix *A* being represented in the array *a* . Its length is equal to the length of the array *a*.

fpm

INTEGER

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

emin, *emax*

REAL for sfeast_scsrev and cfeast_hcsrev

DOUBLE PRECISION for dfeast_scsrev and zfeast_hcsrev

The lower and upper bounds of the interval to be searched for eigenvalues; *emin* ≤ *emax*.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0

INTEGER

On entry, specifies the initial guess for subspace dimension to be used, 0 < *m0*≤*n*. Set *m0* ≥ *m* where *m* is the total number of eigenvalues located in the interval [*emin*, *emax*]. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

x

REAL for sfeast_scsrev

DOUBLE PRECISION for dfeast_scsrev

COMPLEX for cfeast_hcsrev

COMPLEX*16 for zfeast_hcsrev

On entry, if *fpm*(5)=1, the array *x*(*n*, *m*) contains a basis of guess subspace where *n* is the order of the input matrix.

Output Parameters

fpm

On output, the last 64 values correspond to Intel® oneAPI Math Kernel Library (oneMKL) PARDISO*iparm*(1) to *iparm*(64) (regardless of the value of *fpm*(64) on input).

epsout

REAL for sfeast_scsrev and cfeast_hcsrev

DOUBLE PRECISION for dfeast_scsrev and zfeast_hcsrev

On output, contains the relative error on the trace: |$trace_i$ - $trace_{i-1}$| /max(|*emin*|, |*emax*|)

loop

INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

e

REAL for sfeast_scsrev and cfeast_hcsrev

DOUBLE PRECISION for dfeast_scsrev and zfeast_hcsrev

Array of length *m0*. On output, the first *m* entries of *e* are eigenvalues found in the interval.

x

On output, the first *m* columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the *i*-th column of *x* holding the eigenvector associated with *e*(*i*).

m

INTEGER

The total number of eigenvalues found in the interval [*emin*, *emax*]: 0 ≤ *m* ≤ *m0*.

res — `REAL` for `sfeast_scsrev` and `cfeast_hcsrev`

`DOUBLE PRECISION` for `dfeast_scsrev` and `zfeast_hcsrev`

Array of length *m0*. On exit, the first *m* components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max\left(|E_{\min}|, |E_{\max}|\right)\|x_i\|_1}$$

for *i*=1, 2, …, *m*, and where *m* is the total number of eigenvalues found in the search interval.

info — `INTEGER`

If *info*=0, the execution is successful. If *info* ≠ 0, see Output Eigensolver info Details.

?feast_scsrgv/?feast_hcsrgv

Extended Eigensolver interface for generalized eigenvalue problem with sparse matrices.

Syntax

```
call sfeast_scsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call dfeast_scsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call cfeast_hcsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
call zfeast_hcsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

Include Files

- `mkl.fi`

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

NOTE
Both matrices *A* and *B* must use the same family of storage format. The position of the non-zero elements can be different (CSR coordinates *ib* and *jb* can be different from *ia* and *ja*).

Input Parameters

uplo — `CHARACTER*1`

Must be `'U'` or `'L'` or `'F'` .

If *UPLO* = `'U'`, *a* and *b* store the upper triangular parts of *A* and *B* respectively.

If *UPLO* = 'L', *a* and *b* store the lower triangular parts of *A* and *B* respectively.

If *UPLO*= 'F', *a* and *b* store the full matrices *A* and *B* respectively.

n

INTEGER

Sets the size of the problem. *n* > 0.

a

REAL for sfeast_scsrgv

DOUBLE PRECISION for dfeast_scsrgv

COMPLEX for cfeast_hcsrgv

COMPLEX*16 for zfeast_hcsrgv

Array containing the nonzero elements of either the full matrix *A* or the upper or lower triangular part of the matrix *A*, as specified by *uplo*.

ia

INTEGER

Array of length *n* + 1, containing indices of elements in the array *a* , such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i* . The value of the last element *ia*(*n* + 1) is equal to the number of non-zeros plus one.

ja

INTEGER

Array containing the column indices for each non-zero element of the matrix *A* being represented in the array *a* . Its length is equal to the length of the array *a*.

b

REAL for sfeast_scsrgv

DOUBLE PRECISION for dfeast_scsrgv

COMPLEX for cfeast_hcsrgv

COMPLEX*16 for zfeast_hcsrgv

Array of dimension *ldb* by *, contains the nonzero elements of either the full matrix *B* or the upper or lower triangular part of the matrix *B*, as specified by *uplo*.

ib

INTEGER

Array of length *n* + 1, containing indices of elements in the array *b* , such that *ib*(*i*) is the index in the array *b* of the first non-zero element from the row *i* . The value of the last element *ib*(*n* + 1) is equal to the number of non-zeros plus one.

jb

INTEGER

Array containing the column indices for each non-zero element of the matrix *B* being represented in the array *b* . Its length is equal to the length of the array *b*.

fpm

INTEGER

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

emin, *emax*

`REAL` for `sfeast_scsrgv` and `cfeast_hcsrgv`

`DOUBLE PRECISION` for `dfeast_scsrgv` and `zfeast_hcsrgv`

The lower and upper bounds of the interval to be searched for eigenvalues; *emin* ≤ *emax*.

> **NOTE** Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0

`INTEGER`

On entry, specifies the initial guess for subspace dimension to be used, 0 < *m0*≤*n*. Set *m0* ≥ *m* where *m* is the total number of eigenvalues located in the interval [*emin*, *emax*]. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

x

`REAL` for `sfeast_scsrgv`

`DOUBLE PRECISION` for `dfeast_scsrgv`

`COMPLEX` for `cfeast_hcsrgv`

`COMPLEX*16` for `zfeast_hcsrgv`

On entry, if *fpm*(5)=1, the array *x*(*n*, *m*) contains a basis of guess subspace where *n* is the order of the input matrix.

Output Parameters

fpm

On output, the last 64 values correspond to Intel® oneAPI Math Kernel Library (oneMKL) PARDISO*iparm*(1) to *iparm*(64) (regardless of the value of *fpm*(64) on input).

epsout

`REAL` for `sfeast_scsrgv` and `cfeast_hcsrgv`

`DOUBLE PRECISION` for `dfeast_scsrgv` and `zfeast_hcsrgv`

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}|$ /max(|*emin*|, |*emax*|)

loop

`INTEGER`

On output, contains the number of refinement loop executed. Ignored on input.

e

`REAL` for `sfeast_scsrgv` and `cfeast_hcsrgv`

`DOUBLE PRECISION` for `dfeast_scsrgv` and `zfeast_hcsrgv`

Array of length *m0*. On output, the first *m* entries of *e* are eigenvalues found in the interval.

x	On output, the first *m* columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the *i*-th column of *x* holding the eigenvector associated with *e*(*i*).
m	INTEGER The total number of eigenvalues found in the interval [*emin*, *emax*]: 0 ≤ *m* ≤ *m0*.
res	REAL for sfeast_scsrgv and cfeast_hcsrgv DOUBLE PRECISION for dfeast_scsrgv and zfeast_hcsrgv Array of length *m0*. On exit, the first *m* components contain the relative residual vector: $\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(\|E_{\min}\|, \|E_{\max}\|)\|Bx_i\|_1}$ for *i*=1, 2, …, *m*, and where *m* is the total number of eigenvalues found in the search interval.
info	INTEGER If *info*=0, the execution is successful. If *info* ≠ 0, see Output Eigensolver info Details.

Extended Eigensolver Interfaces for Extremal Eigenvalues/Singular Values

The topics in this section discuss Extended Eigensolver interfaces to find extremal eigenvalues as well as singular values.

Extended Eigensolver Interfaces to find largest/smallest eigenvalues

The predefined interfaces include routines for standard and generalized eigenvalue problems and sparse matrices.

Matrix Type	Standard Eigenvalue Problem	Generalized Eigenvalue Problem
Sparse	mkl_sparse_?_ev	mkl_sparse_?_gv

mkl_sparse_?_ev

Computes the largest/smallest eigenvalues and corresponding eigenvectors of a standard eigenvalue problem

Syntax

```
stat = mkl_sparse_s_ev (which, pm, A, descrA, k0, k, E, X, res);
stat = mkl_sparse_d_ev (which, pm, A, descrA, k0, k, E, X, res);
```

Include Files

- mkl_solvers_ee.f90

Description

The `mkl_sparse_?_ev` routine computes the largest/smallest eigenvalues and corresponding eigenvectors of a standard eigenvalue problem.

```
Ax = lambda x
```

where `A` is the real symmetric matrix.

Input Parameters

which — CHARACTER

Indicates eigenvalues for which to search:

- `which = 'L'` indicates the largest eigenvalues.
- `which = 'S'` indicates the smallest eigenvalues.

pm — C_INT

Array of size 128. This array is used to pass various parameters to Extended Eigensolver routines. See • Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem for a complete description of the parameters and their default values.

A — SPARSE_MATRIX_T

Handle containing sparse matrix in internal data structure.

descrA — MATRIX_DESCR

Structure specifying sparse matrix properties.

sparse_matrix_type_t *type* — Specifies the type of a sparse matrix:

- `SPARSE_MATRIX_TYPE_GENERAL`

  The matrix is processed as-is.
- `SPARSE_MATRIX_TYPE_SYMMETRIC`

  The matrix is symmetric (only the requested triangle is processed).

sparse_fill_mode_t *mode* — Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

- `SPARSE_FILL_MODE_LOWER`

  The lower triangular matrix part is processed.
- `SPARSE_FILL_MODE_UPPER`

  The upper triangular matrix part is processed.

sparse_diag_type_t *diag* — Specifies the diagonal type for non-general matrices:

- `SPARSE_DIAG_NON_UNIT`

  Diagonal elements might not be equal to one.
- `SPARSE_DIAG_UNIT`

Diagonal elements are equal to one

k0 — C_INT

The desired number of the largest/smallest eigenvalues to find.

Output Parameters

k — C_INT

Number of eigenvalues found.

E — C_FLOAT for `mkl_sparse_s_ev`

C_DOUBLE for `mkl_sparse_d_ev`

Array of size *k0*. Contains k largest/smallest eigenvalues.

X — C_FLOAT for `mkl_sparse_s_ev`

C_DOUBLE for `mkl_sparse_d_ev`

Array of size *k0**Number of columns of the matrix A. Contains *k* eigenvectors.

Res — C_FLOAT for `mkl_sparse_s_ev`

C_DOUBLE for `mkl_sparse_d_ev`

Array of size *k0*. Contains *k* residuals.

Stat — INTEGER

The function returns a value indicating whether the operation was successful or not, and why.

Return Values

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.
`SPARSE_STATUS_EXECUTION_FAILED`	Execution failed.
`SPARSE_STATUS_INTERNAL_ERROR`	An error in algorithm implementation occurred.
`SPARSE_STATUS_NOT_SUPPORTED`	The requested operation is not supported.

mkl_sparse_?_gv

Computes the largest/smallest eigenvalues and corresponding eigenvectors of a generalized eigenvalue problem

Syntax

```
stat = mkl_sparse_s_gv (which, pm, A, descrA, B, descrB, k0, k, E, X, res);
stat = mkl_sparse_d_gv (which, pm, A, descrA, B, descrB, k0, k, E, X, res);
```

Include Files

- mkl_solvers_ee.f90

Description

The mkl_sparse_?_gv routine computes the largest/smallest eigenvalues and corresponding eigenvectors of a generalized eigenvalue problem.

```
Ax = lambda Bx
```

where A is the real symmetric matrix and B is the real symmetric positive definite matrix.

Input Parameters

which

CHARACTER

Indicates eigenvalues for which to search:

- which = 'L' indicates the largest eigenvalues.
- which = 'S' indicates the smallest eigenvalues.

pm

C_INT

Array of size 128. This array is used to pass various parameters to Extended Eigensolver routines. See • Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem for a complete description of the parameters and their default values.

A

SPARSE_MATRIX_T

Handle containing sparse matrix in internal data structure.

descrA

MATRIX_DESCR

Structure specifying sparse matrix properties.

sparse_matrix_type_t *type*

Specifies the type of a sparse matrix:

- SPARSE_MATRIX_TYPE_GENERAL

  The matrix is processed as-is.
- SPARSE_MATRIX_TYPE_SYMMETRIC

  The matrix is symmetric (only the requested triangle is processed).

sparse_fill_mode_t *mode*

Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

- SPARSE_FILL_MODE_LOWER

  The lower triangular matrix part is processed.
- SPARSE_FILL_MODE_UPPER

  The upper triangular matrix part is processed.

sparse_diag_type_t *diag*

Specifies the diagonal type for non-general matrices:

- SPARSE_DIAG_NON_UNIT

Diagonal elements might not be equal to one.

- SPARSE_DIAG_UNIT

  Diagonal elements are equal to one

B

SPARSE_MATRIX_T

Handle containing sparse matrix in internal data structure.

descrB

MATRIX_DESCR

Structure specifying sparse matrix properties.

sparse_matrix_type_t *type*

Specifies the type of a sparse matrix:

- SPARSE_MATRIX_TYPE_GENERAL

  The matrix is processed as-is.
- SPARSE_MATRIX_TYPE_SYMMETRIC

  The matrix is symmetric (only the requested triangle is processed).

sparse_fill_mode_t *mode*

Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

- SPARSE_FILL_MODE_LOWER

  The lower triangular matrix part is processed.
- SPARSE_FILL_MODE_UPPER

  The upper triangular matrix part is processed.

sparse_diag_type_t *diag*

Specifies the diagonal type for non-general matrices:

- SPARSE_DIAG_NON_UNIT

  Diagonal elements might not be equal to one.
- SPARSE_DIAG_UNIT

  Diagonal elements are equal to one

k0

C_INT

The desired number of the largest/smallest eigenvalues to find.

Output Parameters

k

C_INT

Number of eigenvalues found.

E

C_FLOAT for `mkl_sparse_s_gv`

C_DOUBLE for `mkl_sparse_d_gv`

Array of size *k0*. Contains *k* largest/smallest eigenvalues.

X	C_FLOAT for `mkl_sparse_s_gv` C_DOUBLE for `mkl_sparse_d_gv` Array of size *k0**Number of columns of matrix A. Contains *k* eigenvectors.
Res	C_FLOAT for `mkl_sparse_s_gv` C_DOUBLE for `mkl_sparse_d_gv` Array of size *k0*. Contains *k* residuals.
Stat	INTEGER The function returns a value indicating whether the operation was successful or not, and why.

Return Values

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.
`SPARSE_STATUS_EXECUTION_FAILED`	Execution failed.
`SPARSE_STATUS_INTERNAL_ERROR`	An error in algorithm implementation occurred.
`SPARSE_STATUS_NOT_SUPPORTED`	The requested operation is not supported.

Extended Eigensolver Interfaces to find largest/smallest singular values

The predefined interfaces include routines to find the largest and smallest singular values and the corresponding singular vectors of sparse matrices.

Matrix Type	Standard singular value problem
Sparse	mkl_sparse_?_svd

mkl_sparse_?_svd

Computes the largest/smallest singular values of a singular-value problem

Syntax

```
stat = mkl_sparse_s_svd (whichS, whichV, pm, A, descrA, k0, k, E, XL, XR, res);
stat = mkl_sparse_d_svd (whichS, whichV, pm, A, descrA, k0, k, E, XL, XR, res);
```

Include Files

- `mkl_solvers_ee.f90`

Description

The `mkl_sparse_?_svd` routine computes the largest/smallest singular values of a singular-value problem.

`AATx = σx` or `ATAx = σx`, where A is the real rectangular matrix.

Input Parameters

whichS

CHARACTER

Indicates eigenvalues for which to search:

- `whichS = 'L'` indicates the largest eigenvalues.
- `whichS = 'S'` indicates the smallest eigenvalues.

whichV

CHARACTER

Indicates singular vectors for which to search:

- `whichV = 'R'` indicates right singular vectors.
- `whichV = 'L'` indicates left singular vectors.

pm

C_INT

Array of size 128. This array is used to pass various parameters to Extended Eigensolver routines. See • Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem for a complete description of the parameters and their default values.

A

SPARSE_MATRIX_T

Handle containing sparse matrix in internal data structure.

descrA

MATRIX_DESCR

Structure specifying sparse matrix properties.

sparse_matrix_type_t *type*

Specifies the type of a sparse matrix:

- `SPARSE_MATRIX_TYPE_GENERAL`

  The matrix is processed as-is.
- `SPARSE_MATRIX_TYPE_SYMMETRIC`

  The matrix is symmetric (only the requested triangle is processed).

sparse_fill_mode_t *mode*

Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

- `SPARSE_FILL_MODE_LOWER`

  The lower triangular matrix part is processed.
- `SPARSE_FILL_MODE_UPPER`

  The upper triangular matrix part is processed.

sparse_diag_type_t *diag*

Specifies the diagonal type for non-general matrices:

- `SPARSE_DIAG_NON_UNIT`

  Diagonal elements might not be equal to one.
- `SPARSE_DIAG_UNIT`

  Diagonal elements are equal to one

k0	C_INT The desired number of the largest/smallest eigenvalues to find.

Output Parameters

k	C_INT Number of eigenvalues found.
E	C_FLOAT for `mkl_sparse_s_svd` C_DOUBLE for `mkl_sparse_d_svd` Array of size *k0*. Contains *k* largest/smallest eigenvalues.
XL	Contains *k*-corresponding left singular vectors.
XR	Contains *k*-corresponding right singular vectors.
Res	C_FLOAT for `mkl_sparse_s_svd` C_DOUBLE for `mkl_sparse_d_svd` Array that contains *k* residuals.
Stat	INTEGER The function returns a value indicating whether the operation was successful or not, and why.

Return Values

`SPARSE_STATUS_SUCCESS`	The operation was successful.
`SPARSE_STATUS_NOT_INITIALIZED`	The routine encountered an empty handle or matrix array.
`SPARSE_STATUS_ALLOC_FAILED`	Internal memory allocation failed.
`SPARSE_STATUS_INVALID_VALUE`	The input parameters contain an invalid value.
`SPARSE_STATUS_EXECUTION_FAILED`	Execution failed.
`SPARSE_STATUS_INTERNAL_ERROR`	An error in algorithm implementation occurred.
`SPARSE_STATUS_NOT_SUPPORTED`	The requested operation is not supported.

mkl_sparse_ee_init

Initializes Extended Eigensolver input parameters with default values

Syntax

```
stat = mkl_sparse_ee_init (MKL_INT* pm);
```

Include Files

- `mkl_solvers_ee.f90`

Description

This routine sets all Extended Eigensolver parameters to their default values.

Output Parameters

pm	C_INT Array of size 128. This array is used to pass various parameters to Extended Eigensolver routines. See • Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem for a complete description of the parameters and their default values.

Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem

The input parameters for Extended Eigensolver routines are contained in an integer array named *pm*. To call the Extended Eigensolver interfaces, initialize this array using the `mkl_sparse_ee_init` routine.

Parameter	Default	Description
pm(1)	0	Reserved for future use.
pm(2)	6	Defines the tolerance for the stopping criteria: $\in = 10^{-pm(2)+1}$
pm(3)	0	Specifies the algorithm to use: • 0 - Decided at runtime • 1 - The Krylov-Schur method • 2 - Subspace Iteration technique based on FEAST algorithm
pm(4)	*	This parameter is referenced only for Krylov-Schur Method. It indicates the number of Lanczos/Arnoldi vectors (NCV) generated at each iteration. This parameter must be less than or equal to size of matrix and greater than number of eigenvalues (k0) to be computed. If unspecified, NCV is set to be at least 1.5 times larger than k0.
pm(5)	*	Maximum number of iterations. If unspecified, this parameter is set to 10000 for the Krylov-Schur method and 60 for the subspace iteration method.
pm(6)	0	Power of Chebychev expansion for approximate spectral projector. Only referenced when *pm*(3) =1
pm(7)	1	Used only for Krylov-Schur Method. If 0, then the method only computes eigenvalues. If 1, then the method computes eigenvalues and eigenvectors. The subspace iteration method always computes eigenvectors/singular vectors. You must allocate the required memory space.
pm(8)	0	Convergence stopping criteria. Defines whether the stopping criteria for the iterations with respect to the true residuals (used if pm(9) is not zero) and residual norm estimates are relative to the eigenvalues/singular values or not. If 0, the stopping criteria with respect to the true residuals is: $\frac{\|\|Ax-\lambda x\|\|}{\|\lambda\|} < 10^{-pm(2)+1}$

Parameter	Default	Description
		or $\frac{\|\|Ax-\lambda Bx\|\|}{\|\lambda\|} < 10^{-pm(2)+1}$ If 1, the stopping criteria with respect to the true residuals is: $\left\|\left\|Ax-\lambda x\right\|\right\| < 10^{-pm(2)+1}$ or $\left\|\left\|Ax-\lambda Bx\right\|\right\| < 10^{-pm(2)+1}$ for a generalized eigenproblem. The residual norm estimates are based on the magnitude of the last eigenvector of the Schur decomposition matrix and the exact formula can be found in the literature. When `pm(8)=0`, the residual norm estimate is additionally divided by the magnitude of the computed eigenvalue and compared to $10^{(-pm(2)+1)}$.
pm(9)	0	Specifies if for detecting convergence the solver must compute the true residuals for eigenpairs for the Krylov-Schur method or it can only use the residual norm estimates. If 0, only residual norm estimates are used. If 1, the solver computes not just residual norm estimates but also the true residuals as defined in the description of pm(8).
pm(10)	0	Used only for the Krylov-Schur method and only as an output parameter. Reports the reason for exiting the iteration loop of the method: • If 0, the iterations stopped since convergence has been detected. • If -1, maximum number of iterations has been reached and even the residual norm estimates have not converged. • If -2, maximum number of iterations has been reached despite the residual norm estimates have converged (but the true residuals for eigenpairs have not). • If -3, the iterations stagnated and even the residual norm estimates have not converged. • If -4, the iterations stagnated while the eigenvalues have converged (but the true residuals for eigenpairs do not).
pm(11) to *pm*(129)	-	Reserved for future use.

Vector Mathematical Functions

Intel® oneAPI Math Kernel Library (oneMKL) Vector Mathematics functions (VM) compute a mathematical function of each of the vector elements. VM includes a set of highly optimized functions (arithmetic, power, trigonometric, exponential, hyperbolic, special, and rounding) that operate on vectors of real and complex numbers.

Application programs that improve performance with VM include nonlinear programming software, computation of integrals, financial calculations, computer graphics, and many others.

VM functions fall into the following groups according to the operations they perform:

- VM Mathematical Functions compute values of mathematical functions, such as sine, cosine, exponential, or logarithm, on vectors stored contiguously in memory.
- VM Pack/Unpack Functions convert to and from vectors with positive increment indexing, vector indexing, and mask indexing (see Appendix "Vector Arguments in VM" for details on vector indexing methods).
- VM Service Functions set/get the accuracy modes and the error codes, and free memory.

The VM mathematical functions take an input vector as an argument, compute values of the respective function element-wise, and return the results in an output vector. All the VM mathematical functions can perform in-place operations, where the input and output arrays are at the same memory locations. For VM mathematical functions with positive increment indexing, in-place operations are supported only when the input and output increments have the same value.

The Intel® oneAPI Math Kernel Library (oneMKL) interfaces are given in `mkl_vml.f90`; the `mkl_vml.fi` include file available in the previous versions of Intel® oneAPI Math Kernel Library (oneMKL) is retained for backward compatibility

Examples that demonstrate how to use the VM functions are located in:

`${MKL}/examples/vmlf/source`

See VM performance and accuracy data in the online VM Performance and Accuracy Data document available at https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

VM Data Types, Accuracy Modes, and Performance Tips

VM includes mathematical and pack/unpack vector functions for single and double precision vector arguments of real and compex types. Intel® oneAPI Math Kernel Library (oneMKL) provides Fortran and C interfaces for all VM functions, including the associated service functions. The Function Naming Conventions topic shows how to call these functions.

Performance depends on a number of factors, including vectorization and threading overhead. The recommended usage is as follows:

- Use VM for vector lengths larger than 40 elements.
- Use the Intel® Compiler for vector lengths less than 40 elements.

All VM vector functions support the following accuracy modes:

- High Accuracy (HA), the default mode
- Low Accuracy (LA), which improves performance by reducing accuracy of the two least significant bits
- Enhanced Performance (EP), which provides better performance at the cost of significantly reduced accuracy. Approximately half of the bits in the mantissa are correct.

Note that using the EP mode does not guarantee accurate processing of corner cases and special values. Although the default accuracy is HA, LA is sufficient in most cases. For applications that require less accuracy (for example, media applications, some Monte Carlo simulations, etc.), the EP mode may be sufficient.

VM handles special values in accordance with the C99 standard [C99].

Intel® oneAPI Math Kernel Library (oneMKL) offers both functions and environment variables to switch between modes for VM. See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details about the environment variables. Use the `vmlsetmode(`*`mode`*`)` function (see Table "Values of the *mode* Parameter") to switch between the HA, LA, and EP modes. The `vmlgetmode()` function returns the current mode.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

See Also

Function Naming Conventions

VM Naming Conventions

The VM function names are lowercase.

The VM mathematical and pack/unpack function names have the following structure:

`v[m]<?><name><mod>`

where

- *v* is a prefix indicating vector operations.
- `[m]` is an optional prefix for mathematical functions that indicates additional argument to specify a VM mode for a given function call (see `vmlsetmode` for possible values and their description).
- `<?>` is a precision prefix that indicates one of the following data types:

s	`REAL (KIND=4).`
d	`REAL (KIND=8).`
c	`COMPLEX (KIND=4).`
z	`COMPLEX (KIND=8).`

- `<name>` indicates the function short name. See examples in Table "VM Mathematical Functions".
- *<mod>* field is present only in the pack/unpack functions and indicates the indexing method used:

i	indexing with a positive increment
v	indexing with an index vector
m	indexing with a mask vector.

The VM service function names have the following structure:

vml`<name>`

where

`<name>` indicates the function short name. See examples in Table "VM Service Functions".

To call VM functions from an application program, use conventional function calls. For example, call the vector single precision real exponential function as

`call vsexp ( n, a, y )`

`call vmsexp ( n, a, y, mode )` with a specified mode

VM Function Interfaces

VM interfaces include the function names and argument lists. The following sections describe the interfaces for the VM functions. Note that some of the functions have multiple input and output arguments

Some VM functions may also take scalar arguments as input. See the function description for the naming conventions of such arguments.

VM Mathematical Function Interfaces

```
call v<?><name>( n, a, [scalar input arguments,]y )
call v<?><name>i( n, a, inca, [scalar input arguments,]y, incy )
call v<?><name>( n, a, b, [scalar input arguments,]y )
callv<?><name>i( n, a, inca, b, incb, [scalar input arguments,]y, incy )
call v<?><name>( n, a, y, z )
callv<?><name>i( n, a, inca, y, incy, z, incz )
call vm<?><name>( n, a, [scalar input arguments,]y, mode )
call vm<?><name>i( n, a, inca, [scalar input arguments,]y, incy, mode )
call vm<?><name>( n, a, b, [scalar input arguments,]y, mode )
callvm<?><name>i( n, a, inca, b, incb, [scalar input arguments,]y, incy, mode )
call vm<?><name>( n, a, y, z, mode )
callvm<?><name>i( n, a, inca, y, incy, z, incz, mode )
```

VM Mathematical Functions

VM Pack Function Interfaces

```
callv<?>packi( n, a, inca, y )
call v<?>packv( n, a, ia, y )
call v<?>packm( n, a, ma, y )
```

VM Unpack Function Interfaces

```
call v<?>unpacki( n, a, y, incy )
call v<?>unpackv( n, a, y, iy )
call v<?>unpackm( n, a, y, my )
```

VM Service Function Interfaces

```
oldmode = vmlsetmode( mode )
mode = vmlgetmode( )
olderr = vmlseterrstatus ( err )
err = vmlgeterrstatus( )
olderr = vmlclearerrstatus( )
oldcallback = vmlseterrorcallback( callback )
callback = vmlgeterrorcallback( )
oldcallback = vmlclearerrorcallback( )
```

Note that *oldmode*, *olderr*, and *oldcallback* refer to settings prior to the call.

VM Input Parameters

n	number of elements to be calculated
a	first input vector
b	second input vector
inca	vector increment for the input vector *a*
incb	vector increment for the input vector *b*
ia	index vector for the input vector *a*
ma	mask vector for the input vector *a*

incy	vector increment for the output vector *y*
incz	vector increment for the output vector *z*
iy	index vector for the output vector *y*
my	mask vector for the output vector *y*
err	error code
mode	VM mode
callback	address of the callback function

VM Output Parameters

y	first output vector
z	second output vector
err	error code
mode	VM mode
olderr	former error code
oldmode	former VM mode
callback	address of the callback function
oldcallback	address of the former callback function

See the data types of the parameters used in each function in the respective function description section. All Intel® oneAPI Math Kernel Library (oneMKL) VM mathematical functions can perform in-place operations. For VM mathematical functions with positive increment indexing, (for example, v?PowI), in-place operations are supported only when the input and output increments have the same value.

Vector Indexing Methods

Classic VM mathematical functions work with unit stride. Strided VM mathematical functions (names with "I" suffix) work with arbitrary integer increments. Increments may be positive, negative or equal to zero. For example:

```
VSEXPI (n, a, inca, r, incr)
```

is equivalent to:

```
for (i=0; i<n; i++)
{
    r[i * incr] = exp (a[i * inca]);
}
```

```
do i=1, n
    r((i-1)*incr+1) = EXP (a((i-1)*inca+1))
end do
```

where

i – current index,

inca – input index increment,

incr – output index increment.

n – the number of elements to be computed (important: *n* is not the maximum array size).

So, when calling `VSEXPI(n, a, inca, r, incr)` be sure that the input vector *a* is allocated at least for `1 + (n-1)*inca` elements and the result vector *r* has a space for `1 + (n-1)*incr` elements.

> **NOTE** The order of computations is not guaranteed and no array bounds-checking is performed; therefore, the results for overlapped and in-place arrays are not generally deterministic for increments other than 1.

For output index increment, equal to 0, the result is not deterministic and generally nonsensical.

Use negative increments to step from base pointers in reverse order.

For example:

```
VSEXPI (n, a, -2, r, -3)
```

is equivalent to:

```
do i=1, n
    r((i-1)*(-3)+1)  = EXP (a((i-1)*(-2)+1));
end do
```

> **NOTE** Pass pointers to the desired ending array element in memory as an argument for negative strides.

For example:

```
VSEXPI (n, a, 2, r(1000:), -3)
```

Use a zero increment for one fixed argument rather than an array.

For example:

```
VSMULI (n, a, 1, b, 0, r, 1)
```

is equivalent to:

```
do i=1, n
    r(i) = a(i) * b(1)
end do
```

VM Pack/Unpack functions use the following indexing methods to do this task:

- positive increment
- index vector
- mask vector

The indexing method used in a particular function is indicated by the indexing modifier (see the description of the *<mod>* field in Function Naming Conventions). For more information on the indexing methods, see Vector Arguments in VM.

VM Pack/Unpack Functions

VM Error Diagnostics

The VM mathematical functions incorporate the error handling mechanism, which is controlled by the following service functions:

`vmlgeterrstatus`, `vmlseterrstatus`, `vmlclearerrstatus`	These functions operate with a global variable called VM Error Status. The VM Error Status flags an error, a warning, or a successful execution of a VM function.

vmlgeterrcallback, vmlseterrcallback, vmlclearerrcallback	These functions enable you to customize the error handling. For example, you can identify a particular argument in a vector where an error occurred or that caused a warning.
vmlsetmode, vmlgetmode	These functions get and set a VM mode. If you set a new VM mode using the vmlsetmode function, you can store the previous VM mode returned by the routine and restore it at any point of your application.

If both an error and a warning situation occur during the function call, the VM Error Status variable keeps only the value of the error code. See Table "Values of the VM Error Status" for possible values. If a VM function does not encounter errors or warnings, it sets the VM Error Status to VML_STATUS_OK.

If you use incorrect input arguments to a VM function (VML_STATUS_BADSIZE and VML_STATUS_BADMEM), the function calls xerbla to report errors. See Table "Values of the VM Error Status" for details.

You can use the vmlsetmode and vmlgetmode functions to modify error handling behavior. Depending on the VM mode, the error handling behavior includes the following operations:

- setting the VM Error Status to a value corresponding to the observed error or warning
- writing error text information to the *stderr* stream
- raising the appropriate exception on an error, if necessary
- calling the additional error handler callback function that is set by vmlseterrorcallback.

See Also

vmlgeterrstatus Gets the VM Error Status.

vmlseterrstatus Sets the new VM Error Status according to *err* and stores the previous VM Error Status to *olderr*Sets the global VM Status according to new values and returns the previous VM Status.

vmlclearerrstatus Sets the VM Error Status to VML_STATUS_OK and stores the previous VM Error Status to *olderr*.

vmlseterrorcallback Sets the additional error handler callback function and gets the old callback function.

vmlgeterrorcallback Gets the additional error handler callback function.

vmlclearerrorcallback Deletes the additional error handler callback function and retrieves the former callback function.

vmlgetmode Gets the VM mode.

vmlsetmode Sets a new mode for VM functions according to the *mode* parameter and stores the previous VM mode to *oldmode*.

VM Mathematical Functions

This section describes VM functions that compute values of mathematical functions on real and complex vector arguments.

Each function is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data, as well as a description of the input/output arguments.

The input range of parameters is equal to the mathematical range of the input data type, unless the function description specifies input threshold values, which mark off the precision overflow, as follows:

- *FLT_MAX* denotes the maximum number representable in single precision real data type
- *DBL_MAX* denotes the maximum number representable in double precision real data type

Table "VM Mathematical Functions" lists available mathematical functions and associated data types.

VM Mathematical Functions

Function	Data Types	Description
Arithmetic Functions		

Function	Data Types	Description
v?add	*s*, *d*, *c*, *z*	Adds vector elements
v?sub	*s*, *d*, *c*, *z*	Subtracts vector elements
v?sqr	*s*, *d*	Squares vector elements
v?mul	*s*, *d*, *c*, *z*	Multiplies vector elements
v?mulbyconj	*c*, *z*	Multiplies elements of one vector by conjugated elements of the second vector
v?conj	*c*, *z*	Conjugates vector elements
v?abs	*s*, *d*, *c*, *z*	Computes the absolute value of vector elements
v?arg	*c*, *z*	Computes the argument of vector elements
v?linearfrac	*s*, *d*	Performs linear fraction transformation of vectors
v?fmod	*s*, *d*	Performs element by element computation of the modulus function of vector *a* with respect to vector *b*
v?remainder	*s*, *d*	Performs element by element computation of the remainder function on the elements of vector *a* and the corresponding elements of vector *b*
Power and Root Functions		
v?inv	*s*, *d*	Inverts vector elements
v?div	*s*, *d*, *c*, *z*	Divides elements of one vector by elements of the second vector
v?sqrt	*s*, *d*, *c*, *z*	Computes the square root of vector elements
v?invsqrt	*s*, *d*	Computes the inverse square root of vector elements
v?cbrt	*s*, *d*	Computes the cube root of vector elements
v?invcbrt	*s*, *d*	Computes the inverse cube root of vector elements
v?pow2o3	*s*, *d*	Computes the cube root of the square of each vector element
v?pow3o2	*s*, *d*	Computes the square root of the cube of each vector element
v?pow	*s*, *d*, *c*, *z*	Raises each vector element to the specified power
v?powx	*s*, *d*, *c*, *z*	Raises each vector element to the constant power
v?powr	*s*, *d*	Computes *a* to the power *b* for elements of two vectors, where the elements of vector argument *a* are all non-negative
v?hypot	*s*, *d*	Computes the square root of sum of squares
Exponential and Logarithmic Functions		
v?exp	*s*, *d*, *c*, *z*	Computes the base *e* exponential of vector elements
v?exp2	*s*, *d*	Computes the base 2 exponential of vector elements
v?exp10	*s*, *d*	Computes the base 10 exponential of vector elements
v?expm1	*s*, *d*	Computes the base *e* exponential of vector elements decreased by 1
v?ln	*s*, *d*, *c*, *z*	Computes the natural logarithm of vector elements
v?log2	*s*, *d*	Computes the base 2 logarithm of vector elements
v?log10	*s*, *d*, *c*, *z*	Computes the base 10 logarithm of vector elements
v?log1p	*s*, *d*	Computes the natural logarithm of vector elements that are increased by 1
v?logb	*s*, *d*	Computes the exponents of the elements of input vector *a*
Trigonometric Functions		
v?cos	*s*, *d*, *c*, *z*	Computes the cosine of vector elements
v?sin	*s*, *d*, *c*, *z*	Computes the sine of vector elements
v?sincos	*s*, *d*	Computes the sine and cosine of vector elements
v?cis	*c*, *z*	Computes the complex exponent of vector elements (cosine and sine combined to complex value)
v?tan	*s*, *d*, *c*, *z*	Computes the tangent of vector elements
v?acos	*s*, *d*, *c*, *z*	Computes the inverse cosine of vector elements
v?asin	*s*, *d*, *c*, *z*	Computes the inverse sine of vector elements
v?atan	*s*, *d*, *c*, *z*	Computes the inverse tangent of vector elements
v?atan2	*s*, *d*	Computes the four-quadrant inverse tangent of ratios of the elements of two vectors

Function	Data Types	Description
v?cospi	*s*, *d*	Computes the cosine of vector elements multiplied by *π*
v?sinpi	*s*, *d*	Computes the sine of vector elements multiplied by *π*
v?tanpi	*s*, *d*	Computes the tangent of vector elements multiplied by *π*
v?acospi	*s*, *d*	Computes the inverse cosine of vector elements divided by *π*
v?asinpi	*s*, *d*	Computes the inverse sine of vector elements divided by *π*
v?atanpi	*s*, *d*	Computes the inverse tangent of vector elements divided by *π*
v?atan2pi	*s*, *d*	Computes the four-quadrant inverse tangent of the ratios of the corresponding elementss of two vectors divided by *π*
v?cosd	*s*, *d*	Computes the cosine of vector elements multiplied by *π*/180
v?sind	*s*, *d*	Computes the sine of vector elements multiplied by *π*/180
v?tand	*s*, *d*	Computes the tangent of vector elements multiplied by *π*/180
Hyperbolic Functions		
v?cosh	*s*, *d*, *c*, *z*	Computes the hyperbolic cosine of vector elements
v?sinh	*s*, *d*, *c*, *z*	Computes the hyperbolic sine of vector elements
v?tanh	*s*, *d*, *c*, *z*	Computes the hyperbolic tangent of vector elements
v?acosh	*s*, *d*, *c*, *z*	Computes the inverse hyperbolic cosine of vector elements
v?asinh	*s*, *d*, *c*, *z*	Computes the inverse hyperbolic sine of vector elements
v?atanh	*s*, *d*, *c*, *z*	Computes the inverse hyperbolic tangent of vector elements.
Special Functions		
v?erf	*s*, *d*	Computes the error function value of vector elements
v?erfc	*s*, *d*	Computes the complementary error function value of vector elements
v?cdfnorm	*s*, *d*	Computes the cumulative normal distribution function value of vector elements
v?erfinv	*s*, *d*	Computes the inverse error function value of vector elements
v?erfcinv	*s*, *d*	Computes the inverse complementary error function value of vector elements
v?cdfnorminv	*s*, *d*	Computes the inverse cumulative normal distribution function value of vector elements
v?lgamma	*s*, *d*	Computes the natural logarithm for the absolute value of the gamma function of vector elements
v?tgamma	*s*, *d*	Computes the gamma function of vector elements
v?expint1	*s*, *d*	Computes the exponential integral of vector elements
Rounding Functions		
v?floor	*s*, *d*	Rounds towards minus infinity
v?ceil	*s*, *d*	Rounds towards plus infinity
v?trunc	*s*, *d*	Rounds towards zero infinity
v?round	*s*, *d*	Rounds to nearest integer
v?nearbyint	*s*, *d*	Rounds according to current mode
v?rint	*s*, *d*	Rounds according to current mode and raising inexact result exception
v?modf	*s*, *d*	Computes the integer and fractional parts
v?frac	*s*, *d*	Computes the fractional part
Miscellaneous Functions		
v?copysign	*s*, *d*	Returns vector of elements of one argument with signs changed to match other argument elements
v?nextafter	*s*, *d*	Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector
v?fdim	*s*, *d*	Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise
v?fmax	*s*, *d*	Returns the larger of each pair of elements of the two vector arguments

Function	Data Types	Description
v?fmin	*s*, *d*	Returns the smaller of each pair of elements of the two vector arguments
v?maxmag	*s*, *d*	Returns the element with the larger magnitude between each pair of elements of the two vector arguments
v?minmag	*s*, *d*	Returns the element with the smaller magnitude between each pair of elements of the two vector arguments

Special Value Notations

This topic defines notations of special values for complex functions. The definitions are provided in text, tables, or formulas.

- `z`, `z1`, `z2`, etc. denote complex numbers.
- `i`, $i^2=-1$ is the imaginary unit.
- `x`, `X`, `x1`, `x2`, etc. denote real imaginary parts.
- `y`, `Y`, `y1`, `y2`, etc. denote imaginary parts.
- `X` and `Y` represent any finite positive IEEE-754 floating point values, if not stated otherwise.
- Quiet `NaN` and signaling `NaN` are denoted with `QNAN` and `SNAN`, respectively.
- The IEEE-754 positive infinities or floating-point numbers are denoted with a `+` sign before `X`, `Y`, etc.
- The IEEE-754 negative infinities or floating-point numbers are denoted with a – sign before `X`, `Y`, etc.

`CONJ(z)` and `CIS(z)` are defined as follows:

`CONJ(x+i·y)=x–i·y`

`CIS(y)=cos(y)+i·sin(y).`

The special value tables show the result of the function for the `z` argument at the intersection of the `RE(z)` column and the `i*IM(z)` row. If the function raises an exception on the argument `z`, the lower part of this cell shows the raised exception and the VM Error Status. An empty cell indicates that this argument is normal and the result is defined mathematically.

Arithmetic Functions

Arithmetic functions perform the basic mathematical operations like addition, subtraction, multiplication or computation of the absolute value of the vector elements.

v?Add

Performs element by element addition of vector a and vector b.

Syntax

```
call vsadd( n, a, b, y )
call vsaddi(n, a, inca, b, incb, y, incy)
call vmsadd( n, a, b, y, mode )
call vmsaddi(n, a, inca, b, incb, y, incy, mode)
call vdadd( n, a, b, y )
call vdaddi(n, a, inca, b, incb, y, incy)
call vmdadd( n, a, b, y, mode )
call vmdaddi(n, a, inca, b, incb, y, incy, mode)
call vcadd( n, a, b, y )
call vcaddi(n, a, inca, b, incb, y, incy)
```

```
call vmcadd( n, a, b, y, mode )
call vmcaddi(n, a, inca, b, incb, y, incy, mode)
call vzadd( n, a, b, y )
call vzaddi(n, a, inca, b, incb, y, incy)
call vmzadd( n, a, b, y, mode )
call vmzaddi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a, *b*	DOUBLE PRECISION for vdadd, vmdadd COMPLEX for vcadd, vmcadd DOUBLE COMPLEX for vzadd, vmzadd REAL, INTENT(IN) for vsAdd, vmsAdd DOUBLE PRECISION, INTENT(IN) for vdadd, vmdadd COMPLEX, INTENT(IN) for vcadd, vmcadd DOUBLE COMPLEX, INTENT(IN) for vzadd, vmzadd	Arrays that specify the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdadd, vmdadd COMPLEX, for vcadd, vmcadd DOUBLE COMPLEX for vzadd, vmzadd REAL, INTENT(OUT) for vsAdd, vmsAdd	Array that specifies the output vector *y*.

Name	Type	Description
	DOUBLE PRECISION, INTENT(OUT) for vdadd, vmdadd	
	COMPLEX, INTENT(OUT) for vcadd, vmcadd	
	DOUBLE COMPLEX, INTENT(OUT) for vzadd, vmzadd	

Description

The v?Add function performs element by element addition of vector *a* and vector *b*.

Special values for Real Function v?Add

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
+∞	+∞	+∞	
+∞	-∞	QNAN	INVALID
-∞	+∞	QNAN	INVALID
-∞	-∞	-∞	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

```
Add(x1+i*y1,x2+i*y2) = (x1+x2) + i*(y1+y2)
```

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to VML_STATUS_OVERFLOW (overriding any possible VML_STATUS_ACCURACYWARNING status).

v?Sub

Performs element by element subtraction of vector b from vector a.

Syntax

```
call vssub( n, a, b, y )
call vssubi(n, a, inca, b, incb, y, incy)
call vmssub( n, a, b, y, mode )
call vmssubi(n, a, inca, b, incb, y, incy, mode)
call vdsub( n, a, b, y )
call vdsubi(n, a, inca, b, incb, y, incy)
call vmdsub( n, a, b, y, mode )
```

```
call vmdsubi(n, a, inca, b, incb, y, incy, mode)
call vcsub( n, a, b, y )
call vcsubi(n, a, inca, b, incb, y, incy)
call vmcsub( n, a, b, y, mode )
call vmcsubi(n, a, inca, b, incb, y, incy, mode)
call vzsub( n, a, b, y )
call vzsubi(n, a, inca, b, incb, y, incy)
call vmzsub( n, a, b, y, mode )
call vmzsubi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a, *b*	DOUBLE PRECISION for vdsub, vmdsub COMPLEX for vcsub, vmcsub DOUBLE COMPLEX for vzsub, vmzsub REAL, INTENT(IN) for vssub, vmssub DOUBLE PRECISION, INTENT(IN) for vdsub, vmdsub COMPLEX, INTENT(IN) for vcsub, vmcsub DOUBLE COMPLEX, INTENT(IN) for vzsub, vmzsub	Arrays that specify the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vssub, vmssub DOUBLE PRECISION for vdsub, vmdsub	Array that specifies the output vector *y*.

Name	Type	Description
	COMPLEX for vcsub, vmcsub	
	DOUBLE COMPLEX for vzsub, vmzsub	
	REALINTENT(OUT) for vssub, vmssub	
	DOUBLE PRECISION, INTENT(OUT) for vdsub, vmdsub	
	COMPLEX, INTENT(OUT) for vcsub, vmcsub	
	DOUBLE COMPLEX, INTENT(OUT) for vzsub, vmzsub	

Description

The v?Sub function performs element by element subtraction of vector *b* from vector *a*.

Special values for Real Function v?Sub(x)

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	+0	
-0	+0	-0	
-0	-0	+0	
+∞	+∞	QNAN	INVALID
+∞	-∞	+∞	
-∞	+∞	-∞	
-∞	-∞	QNAN	INVALID
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

Sub(x1+i*y1,x2+i*y2) = (x1-x2) + i*(y1-y2).

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to VML_STATUS_OVERFLOW (overriding any possible VML_STATUS_ACCURACYWARNING status).

v?Sqr

Performs element by element squaring of the vector.

Syntax

```
call vssqr( n, a, y )
call vssqri(n, a, inca, y, incy)
call vmssqr( n, a, y, mode )
call vmssqri(n, a, inca, y, incy, mode)
```

```
call vdsqr( n, a, y )
call vdsqri(n, a, inca, y, incy)
call vmdsqr( n, a, y, mode )
call vmdsqri(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdsqr, vmdsqr REAL, INTENT(IN) for vssqr, vmssqr DOUBLE PRECISION, INTENT(IN) for vdsqr, vmdsqr	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdsqr, vmdsqr REAL, INTENT(OUT) for vssqr, vmssqr DOUBLE PRECISION, INTENT(OUT) for vdsqr, vmdsqr	Array that specifies the output vector *y*.

Description

The v?Sqr function performs element by element squaring of the vector.

Special Values for Real Function v?Sqr(x)

Argument	Result	Exception
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

v?Mul

Performs element by element multiplication of vector a and vector b.

Syntax

```
call vsmul( n, a, b, y )
call vsmuli(n, a, inca, b, incb, y, incy)
call vmsmul( n, a, b, y, mode )
call vmsmuli(n, a, inca, b, incb, y, incy, mode)
call vdmul( n, a, b, y )
call vdmuli(n, a, inca, b, incb, y, incy)
call vmdmul( n, a, b, y, mode )
call vmdmuli(n, a, inca, b, incb, y, incy, mode)
call vcmul( n, a, b, y )
call vcmuli(n, a, inca, b, incb, y, incy)
call vmcmul( n, a, b, y, mode )
call vmcmuli(n, a, inca, b, incb, y, incy, mode)
call vzmul( n, a, b, y )
call vzmuli(n, a, inca, b, incb, y, incy)
call vmzmul( n, a, b, y, mode )
call vmzmuli(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a, *b*	DOUBLE PRECISION for vdmul, vmdmul COMPLEX for vcmul, vmcmul DOUBLE COMPLEX for vzmul, vmzmul REAL, INTENT(IN) for vsmul, vmsmul DOUBLE PRECISION, INTENT(IN) for vdmul, vmdmul COMPLEX, INTENT(IN) for vcmul, vmcmul DOUBLE COMPLEX, INTENT(IN) for vzmul, vmzmul	Arrays that specify the input vectors *a* and *b*.

Name	Type	Description
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdmul, vmdmul COMPLEX, for vcmul, vmcmul DOUBLE COMPLEX for vzmul, vmzmul REAL, INTENT(OUT) for vsmul, vmsmul DOUBLE PRECISION, INTENT(OUT) for vdmul, vmdmul COMPLEX, INTENT(OUT) for vcmul, vmcmul DOUBLE COMPLEX, INTENT(OUT) for vzmul, vmzmul	Array that specifies the output vector *y*.

Description

The v?Mul function performs element by element multiplication of vector *a* and vector *b*.

Special values for Real Function v?Mul(x)

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	-0	
-0	+0	-0	
-0	-0	+0	
+0	+∞	QNAN	INVALID
+0	-∞	QNAN	INVALID
-0	+∞	QNAN	INVALID
-0	-∞	QNAN	INVALID
+∞	+0	QNAN	INVALID
+∞	-0	QNAN	INVALID
-∞	+0	QNAN	INVALID
-∞	-0	QNAN	INVALID
+∞	+∞	+∞	
+∞	-∞	-∞	
-∞	+∞	-∞	
-∞	-∞	+∞	
SNAN	any value	QNAN	INVALID

Argument 1	Argument 2	Result	Exception
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`Mul(x1+i*y1,x2+i*y2) = (x1*x2-y1*y2) + i*(x1*y2+y1*x2).`

Overflow in a complex function occurs (supported in the `HA`/`LA` accuracy modes only) when all `RE(x)`, `RE(y)`, `IM(x)`, `IM(y)` arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?MulByConj

Performs element by element multiplication of vector a element and conjugated vector b element.

Syntax

```
call vcmulbyconj( n, a, b, y )
call vsmulbyconji(n, a, inca, b, incb, y, incy)
call vmcmulbyconj( n, a, b, y, mode )
call vmsmulbyconji(n, a, inca, b, incb, y, incy, mode)
call vzmulbyconj( n, a, b, y )
call vdmulbyconji(n, a, inca, b, incb, y, incy)
call vmzmulbyconj( n, a, b, y, mode )
call vmdmulbyconji(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER, INTENT(IN)`	Specifies the number of elements to be calculated.

a, *b*	`DOUBLE COMPLEX` for `vzmulbyconj`, `vmzmulbyconj`
	`COMPLEX, INTENT(IN)` for `vcmulbyconj`, `vmcmulbyconj`
	`DOUBLE COMPLEX, INTENT(IN)` for `vzmulbyconj`, `vmzmulbyconj`
		Arrays that specify the input vectors *a* and *b*.

inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.

Name	Type	Description
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE COMPLEX for vzmulbyconj, vmzmulbyconj COMPLEX, INTENT(OUT) for vcmulbyconj, vmcmulbyconj DOUBLE COMPLEX, INTENT(OUT) for vzmulbyconj, vmzmulbyconj	Array that specifies the output vector *y*.

Description

The v?MulByConj function performs element by element multiplication of vector *a* element and conjugated vector *b* element.

Specifications for special values of the functions are found according to the formula

MulByConj(x1+i*y1,x2+i*y2) = Mul(x1+i*y1,x2-i*y2).

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to VML_STATUS_OVERFLOW (overriding any possible VML_STATUS_ACCURACYWARNING status).

v?Conj

Performs element by element conjugation of the vector.

Syntax

```
call vcconj( n, a, y )
call vcconji(n, a, inca, y, incy)
call vmcconj( n, a, y, mode )
call vmcconji(n, a, inca, y, incy, mode)
call vzconj( n, a, y )
call vzconji(n, a, inca, y, incy)
call vmzconj( n, a, y, mode )
call vmzconji(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE COMPLEX, INTENT(IN) for vzconj, vmzconj COMPLEX, INTENT(IN) for vcconj, vmcconj DOUBLE COMPLEX, INTENT(IN) for vzconj, vmzconj	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE COMPLEX for vzconj, vmzconj COMPLEX, INTENT(OUT) for vcconj, vmcconj DOUBLE COMPLEX, INTENT(OUT) for vzconj, vmzconj	Array that specifies the output vector *y*.

Description

The v?Conj function performs element by element conjugation of the vector.

No special values are specified. The function does not raise floating-point exceptions.

v?Abs

Computes absolute value of vector elements.

Syntax

```
call vsabs( n, a, y )
call vsabsi(n, a, inca, y, incy)
call vmsabs( n, a, y, mode )
call vmsabsi(n, a, inca, y, incy, mode)
call vdabs( n, a, y )
call vdabsi(n, a, inca, y, incy)
call vmdabs( n, a, y, mode )
call vmdabsi(n, a, inca, y, incy, mode)
call vcabs( n, a, y )
```

```
call vcabsi(n, a, inca, y, incy)
call vmcabs( n, a, y, mode )
call vmcabsi(n, a, inca, y, incy, mode)
call vzabs( n, a, y )
call vzabsi(n, a, inca, y, incy)
call vmzabs( n, a, y, mode )
call vmzabsi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdabs, vmdabs COMPLEX for vcabs, vmcabs DOUBLE COMPLEX for vzabs, vmzabs REAL, INTENT(IN) for vsabs, vmsabs DOUBLE PRECISION, INTENT(IN) for vdabs, vmdabs COMPLEX, INTENT(IN) for vcabs, vmcabs DOUBLE COMPLEX, INTENT(IN) for vzabs, vmzabs	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT (IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdabs, vmdabs, vzabs, vmzabs REAL, INTENT(OUT) for vsabs, vmsabs, vcabs, vmcabs DOUBLE PRECISION, INTENT(OUT) for vdabs, vmdabs, vzabs, vmzabs	Array that specifies the output vector *y*.

Description

The v?Abs function computes an absolute value of vector elements.

Special Values for Real Function v?Abs(x)

Argument	Result	Exception
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

Specifications for special values of the complex functions are defined according to the following formula

Abs(z) = Hypot(RE(z),IM(z)).

v?Arg

Computes argument of vector elements.

Syntax

```
call vcarg( n, a, y )
call vcargi(n, a, inca, y, incy)
call vmcarg( n, a, y, mode )
call vmcargi(n, a, inca, y, incy, mode)
call vzarg( n, a, y )
call vzargi(n, a, inca, y, incy)
call vmzarg( n, a, y, mode )
call vmzargi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	COMPLEX for vcarg, vmcarg DOUBLE COMPLEX for vzarg, vmzarg COMPLEX, INTENT(IN) for vcarg, vmcarg DOUBLE COMPLEX, INTENT(IN) for vzarg, vmzarg	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.

Name	Type	Description
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vzarg, vmzarg REAL, INTENT(OUT) for vcarg, vmcarg DOUBLE PRECISION, INTENT(OUT) for vzarg, vmzarg	Array that specifies the output vector *y*.

Description

The v?Arg function computes argument of vector elements.

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Arg(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+3·π/4	+π/2	+π/2	+π/2	+π/2	+π/4	NAN
+i·Y	+π		+π/2	+π/2		+0	NAN
+i·0	+π	+π	+π	+0	+0	+0	NAN
-i·0	-π	-π	-π	-0	-0	-0	NAN
-i·Y	-π		-π/2	-π/2		-0	NAN
-i·∞	-3·π/4	-π/2	-π/2	-π/2	-π/2	-π/4	NAN
+i·NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN
- Arg(z)=Atan2(IM(z),RE(z)).

v?LinearFrac

Performs linear fraction transformation of vectors a and b with scalar parameters.

Syntax

```
call vslinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y )
call vslinearfraci(n, a, inca, b, incb, scalea, shifta, scaleb, shiftb, y, incy)
call vmslinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode )
call vmslinearfraci(n, a, inca, b, incb, scalea, shifta, scaleb, shiftb, y, incy, mode)
call vdlinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y )
```

```
call vdlinearfraci(n, a, inca, b, incb, scalea, shifta, scaleb, shiftb, y, incy)
call vmdlinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode )
call vmdlinearfraci(n, a, inca, b, incb, scalea, shifta, scaleb, shiftb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a, *b*	DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac REAL, INTENT(IN) for vslinearfrac, vmslinearfrac DOUBLE PRECISION, INTENT(IN) for vdlinearfrac, vmdlinearfrac	Arrays that specify the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
scalea, *scaleb*	DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac REAL, INTENT(IN) for vslinearfrac, vmslinearfrac DOUBLE PRECISION, INTENT(IN) for vdlinearfrac, vmdlinearfrac	Constant values for scaling multipliers of vectors *a* and *b*.
shifta, *shiftb*	DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac REAL, INTENT(IN) for vslinearfrac, vmslinearfrac DOUBLE PRECISION, INTENT(IN) for vdlinearfrac, vmdlinearfrac	Constant values for shifting addends of vectors *a* and *b*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac REAL, INTENT(OUT) for vslinearfrac, vmslinearfrac	Array that specifies the output vector *y*.

Name	Type	Description
	DOUBLE PRECISION, INTENT(OUT) for vdlinearfrac, vmdlinearfrac	

Description

The v?LinearFrac function performs a linear fraction transformation of vector *a* by vector *b* with scalar parameters: scaling multipliers *scalea*, *scaleb* and shifting addends *shifta*, *shiftb*:

$y[i]=(scalea \cdot a[i]+shifta)/(scaleb \cdot b[i]+shiftb)$, $i=1,2 \ldots n$

The v?LinearFrac function is implemented in the EP accuracy mode only, therefore no special values are defined for this function. If used in HA or LA mode, v?LinearFrac sets the VM Error Status to VML_STATUS_ACCURACYWARNING (see the Values of the VM Status table). Correctness is guaranteed within the threshold limitations defined for each input parameter (see the table below); otherwise, the behavior is unspecified.

Threshold Limitations on Input Parameters
$2^{E_{MIN}/2} \le \lvert scalea \rvert \le 2^{(E_{MAX}-2)/2}$
$2^{E_{MIN}/2} \le \lvert scaleb \rvert \le 2^{(E_{MAX}-2)/2}$
$\lvert shifta \rvert \le 2^{E_{MAX}-2}$
$\lvert shiftb \rvert \le 2^{E_{MAX}-2}$
$2^{E_{MIN}/2} \le a[i] \le 2^{(E_{MAX}-2)/2}$
$2^{E_{MIN}/2} \le b[i] \le 2^{(E_{MAX}-2)/2}$
$a[i] \neq -(shifta/scalea)*(1-\delta_1), \lvert\delta_1\rvert \le 2^{1-(p-1)/2}$
$b[i] \neq -(shiftb/scaleb)*(1-\delta_2), \lvert\delta_2\rvert \le 2^{1-(p-1)/2}$

E_{MIN} and E_{MAX} are the minimum and maximum exponents and *p* is the number of significant bits (precision) for the corresponding data type according to the ANSI/IEEE Standard 754-2008 ([IEEE754]):

- for single precision E_{MIN} = -126, E_{MAX} = 127, p = 24
- for double precision E_{MIN} = -1022, E_{MAX} = 1023, p = 53

The thresholds become less strict for common cases with *scalea*=0 and/or *scaleb*=0:

- if *scalea*=0, there are no limitations for the values of *a*[i] and *shifta*.
- if *scaleb*=0, there are no limitations for the values of *b*[i] and *shiftb*.

Example

To use the v?linearfrac to shift vector *a* by a scalar value, set *scaleb* to 0. Note that even if *scaleb* is 0, *b* must be declared.

To use the v?linearfrac to compute *shifta*/(*scaleb*·*b*[*i*]+*shiftb*), set *scalea* to 0. Note that even if *scalea* is 0, *a* must be declared.

v?Fmod

The v?Fmod *function performs element by element computation of the modulus function of vector a with respect to vector b.*

Syntax

```
call vsfmod (n, a, b, y )
call vsfmodi(n, a, inca, b, incb, y, incy)
call vmsfmod (n, a, b, y, mode )
call vmsfmodi(n, a, inca, b, incb, y, incy, mode)
call vdfmod (n, a, b, y )
call vdfmodi(n, a, inca, b, incb, y, incy)
call vmdfmod (n, a, b, y, mode )
call vmdfmodi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a, b	REAL for vsFmod REAL for vmsFmod DOUBLE PRECISION for vdFmod DOUBLE PRECISION for vmdFmod	Pointers to arrays containing the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vsFmod REAL for vmsFmod DOUBLE PRECISION for vdFmod DOUBLE PRECISION for vmdFmod	Pointer to an array containing the output vector *y*.

Description

The v?Fmod function computes the modulus function of each element of vector *a*, with respect to the corresponding elements of vector *b*:

$a_i - b_i * \mathrm{trunc}(a_i/b_i)$

In general, the modulus function fmod (a_i, b_i) returns the value $a_i - n*b_i$ for some integer *n* such that if b_i is nonzero, the result has the same sign as a_i and a magnitude less than the magnitude of b_i.

Special values for Real Function v?Fmod(x, y)

Argument 1	Argument 2	Result	VM Error Status	Exception
x not NAN	±0	NAN	VML_STATUS_SING	INVALID
±∞	*y* not NAN	NAN	VML_STATUS_SING	INVALID
±0	*y*≠ 0, not NAN	±0		
x finite	±∞	*x*		UNDERFLOW if *x* is subnormal
NAN	*y*	NAN		
x	NAN	NAN		

NOTE
If element *i* in the result of `v?Fmod` is 0, its sign is that of a_i.

See Also

Div Performs element by element division of vector *a* by vector *b*

Remainder Performs element by element computation of the remainder function on the elements of vector *a* and the corresponding elements of vector *b*.

v?Remainder

Performs element by element computation of the remainder function on the elements of vector a and the corresponding elements of vector b.

Syntax

```
call vsremainder (n, a, b, y )
call vsremainderi(n, a, inca, b, incb, y, incy)
call vmsremainder (n, a, b, y, mode )
call vmsremainderi(n, a, inca, b, incb, y, incy, mode)
call vdremainder (n, a, b, y )
call vdremainderi(n, a, inca, b, incb, y, incy)
call vmdremainder (n, a, b, y, mode )
call vmdremainderi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER`	Specifies the number of elements to be calculated.

a, b	`REAL` for `vsRemainder`
	`REAL` for `vmsRemainder`
	`DOUBLE PRECISION` for `vdRemainder`
		Pointers to arrays containing the input vectors *a* and *b*.

Name	Type	Description
	DOUBLE PRECISION for vmdRemainder	
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vsRemainder REAL for vmsRemainder DOUBLE PRECISION for vdRemainder DOUBLE PRECISION for vmdRemainder	Pointer to an array containing the output vector *y*.

Description

Computes the remainder of each element of vector *a*, with respect to the corresponding elements of vector *b*: compute the values of *n* such that

$n = a_i - n*b_i$

where *n* is the integer nearest to the exact value of a_i/b_i. If two integers are equally close to a_i/b_i, *n* is the even one. If *n* is zero, it has the same sign as a_i.

Special values for Real Function v?Remainder(x, y)

Argument 1	Argument 2	Result	VM Error Status	Exception
x not NAN	±0	NAN	VML_STATUS_DOM	INVALID
±∞	*y* not NAN	NAN		INVALID
±0	$y \neq 0$, not NAN	±0		
x finite	±∞	*x*		UNDERFLOW if *x* is subnormal
NAN	*y*	NAN		
x	NAN	NAN		

NOTE
If element *i* in the result of v?Remainder is 0, its sign is that of a_i.

See Also

Div Performs element by element division of vector *a* by vector *b*

Fmod The v?Fmod function performs element by element computation of the modulus function of vector *a* with respect to vector *b*.

Power and Root Functions

v?Inv

Performs element by element inversion of the vector.

Syntax

```
call vsinv( n, a, y )
call vsinvi(n, a, inca, y, incy)
call vmsinv( n, a, y, mode )
call vmsinvi(n, a, inca, y, incy, mode)
call vdinv( n, a, y )
call vdinvi(n, a, inca, y, incy)
call vmdinv( n, a, y, mode )
call vmdinvi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdinv, vmdinv REAL, INTENT(IN) for vsinv, vmsinv DOUBLE PRECISION, INTENT(IN) for vdinv, vmdinv	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdinv, vmdinv REAL, INTENT(OUT) for vsinv, vmsinv DOUBLE PRECISION, INTENT(OUT) for vdinv, vmdinv	Array that specifies the output vector *y*.

Description

The v?Inv function performs element by element inversion of the vector.

Special Values for Real Function v?Inv(x)

Argument	Result	VM Error Status	Exception
+0	+∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
+∞	+0		
-∞	-0		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Div

Performs element by element division of vector a by vector b

Syntax

```
call vsdiv( n, a, b, y )
call vsdivi(n, a, inca, b, incb, y, incy)
call vmsdiv( n, a, b, y, mode )
call vmsdivi(n, a, inca, b, incb, y, incy, mode)
call vddiv( n, a, b, y )
call vddivi(n, a, inca, b, incb, y, incy)
call vmddiv( n, a, b, y, mode )
call vmddivi(n, a, inca, b, incb, y, incy, mode)
call vcdiv( n, a, b, y )
call vcdivi(n, a, inca, b, incb, y, incy)
call vmcdiv( n, a, b, y, mode )
call vmcdivi(n, a, inca, b, incb, y, incy, mode)
call vzdiv( n, a, b, y )
call vzdivi(n, a, inca, b, incb, y, incy)
call vmzdiv( n, a, b, y, mode )
call vmzdivi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a, *b*	DOUBLE PRECISION for vddiv, vmddiv COMPLEX for vcdiv, vmcdiv	Arrays that specify the input vectors *a* and *b*.

Name	Type	Description
	DOUBLE COMPLEX for vzdiv, vmzdiv REAL, INTENT(IN) for vsdiv, vmsdiv DOUBLE PRECISION, INTENT(IN) for vddiv, vmddiv COMPLEX, INTENT(IN) for vcdiv, vmcdiv DOUBLE COMPLEX, INTENT(IN) for vzdiv, vmzdiv	
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Div Function

Data Type	Threshold Limitations on Input Parameters
single precision	abs(*a*[i]) < abs(*b*[i]) * FLT_MAX
double precision	abs(*a*[i]) < abs(*b*[i]) * DBL_MAX

Precision overflow thresholds for the complex v?Div function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vddiv, vmddiv COMPLEX for vcdiv, vmcdiv DOUBLE COMPLEX for vzdiv, vmzdiv REAL, INTENT(OUT) for vsdiv, vmsdiv DOUBLE PRECISION, INTENT(OUT) for vddiv, vmddiv COMPLEX, INTENT(OUT) for vcdiv, vmcdiv DOUBLE COMPLEX, INTENT(OUT) for vzdiv, vmzdiv	Array that specifies the output vector *y*.

Description

The v?Div function performs element by element division of vector *a* by vector *b*.

Special values for Real Function v?Div(x)

Argument 1	Argument 2	Result	VM Error Status	Exception
X > +0	+0	+∞	VML_STATUS_SING	ZERODIVIDE

Argument 1	Argument 2	Result	VM Error Status	Exception
X > +0	-0	-∞	VML_STATUS_SING	ZERODIVIDE
X < +0	+0	-∞	VML_STATUS_SING	ZERODIVIDE
X < +0	-0	+∞	VML_STATUS_SING	ZERODIVIDE
+0	+0	QNAN	VML_STATUS_SING	
-0	-0	QNAN	VML_STATUS_SING	
X > +0	+∞	+0		
X > +0	-∞	-0		
+∞	+∞	QNAN		
-∞	-∞	QNAN		
QNAN	QNAN	QNAN		
SNAN	SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

`Div(x1+i*y1,x2+i*y2) = (x1+i*y1)*(x2-i*y2)/(x2*x2+y2*y2).`

Overflow in a complex function occurs when `x2+i*y2` is not zero, `x1`, `x2`, `y1`, `y2` are finite numbers, but the real or imaginary part of the exact result is so large that it does not fit the target precision. In that case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW`.

v?Sqrt

Computes a square root of vector elements.

Syntax

```
call vssqrt( n, a, y )
call vssqrti(n, a, inca, y, incy)
call vmssqrt( n, a, y, mode )
call vmssqrti(n, a, inca, y, incy, mode)
call vdsqrt( n, a, y )
call vdsqrti(n, a, inca, y, incy)
call vmdsqrt( n, a, y, mode )
call vmdsqrti(n, a, inca, y, incy, mode)
call vcsqrt( n, a, y )
call vcsqrti(n, a, inca, y, incy)
call vmcsqrt( n, a, y, mode )
call vmcsqrti(n, a, inca, y, incy, mode)
call vzsqrt( n, a, y )
call vzsqrti(n, a, inca, y, incy)
call vmzsqrt( n, a, y, mode )
call vmzsqrti(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdsqrt, vmdsqrt COMPLEX for vcsqrt, vmcsqrt DOUBLE COMPLEX for vzsqrt, vmzsqrt REAL, INTENT(IN) for vssqrt, vmssqrt DOUBLE PRECISION, INTENT(IN) for vdsqrt, vmdsqrt COMPLEX, INTENT(IN) for vcsqrt, vmcsqrt DOUBLE COMPLEX, INTENT(IN) for vzsqrt, vmzsqrt	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vssqrt, vmssqrt DOUBLE PRECISION for vdsqrt, vmdsqrt COMPLEX for vcsqrt, vmcsqrt DOUBLE COMPLEX for vzsqrt, vmzsqrt REAL, INTENT(OUT) for vssqrt, vmssqrt DOUBLE PRECISION, INTENT(OUT) for vdsqrt, vmdsqrt COMPLEX, INTENT(OUT) for vcsqrt, vmcsqrt DOUBLE COMPLEX, INTENT(OUT) for vzsqrt, vmzsqrt	Array that specifies the output vector *y*.

Description

The `v?Sqrt` function computes a square root of vector elements.

Special Values for Real Function v?Sqrt(x)

Argument	Result	VM Error Status	Exception
X < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+0		
-0	-0		
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Sqrt(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞
+i·Y	+0+i·∞					+∞+i·0	QNAN+i·QNAN
+i·0	+0+i·∞		+0+i·0	+0+i·0		+∞+i·0	QNAN+i·QNAN
-i·0	+0-i·∞		+0-i·0	+0-i·0		+∞-i·0	QNAN+i·QNAN
-i·Y	+0-i·∞					+∞-i·0	QNAN+i·QNAN
-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞
+i·NAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when the real or imaginary part of the argument is `SNAN`
- `Sqrt(CONJ(z))=CONJ(Sqrt(z))`.

v?InvSqrt

Computes an inverse square root of vector elements.

Syntax

```
call vsinvsqrt( n, a, y )
call vsinvsqrti(n, a, inca, y, incy)
call vmsinvsqrt( n, a, y, mode )
call vmsinvsqrti(n, a, inca, y, incy, mode)
call vdinvsqrt( n, a, y )
call vdinvsqrti(n, a, inca, y, incy)
call vmdinvsqrt( n, a, y, mode )
call vmdinvsqrti(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdinvsqrt, vmdinvsqrt REAL, INTENT(IN) for vsinvsqrt, vmsinvsqrt DOUBLE PRECISION, INTENT(IN) for vdinvsqrt, vmdinvsqrt	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdinvsqrt, vmdinvsqrt REAL, INTENT(OUT) for vsinvsqrt, vmsinvsqrt DOUBLE PRECISION, INTENT(OUT) for vdinvsqrt, vmdinvsqrt	Array that specifies the output vector *y*.

Description

The v?InvSqrt function computes an inverse square root of vector elements.

Special Values for Real Function v?InvSqrt(x)

Argument	Result	VM Error Status	Exception
X < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Cbrt

Computes a cube root of vector elements.

Syntax

```
call vscbrt( n, a, y )
call vscbrti(n, a, inca, y, incy)
call vmscbrt( n, a, y, mode )
call vmscbrti(n, a, inca, y, incy, mode)
call vdcbrt( n, a, y )
call vdcbrti(n, a, inca, y, incy)
call vmdcbrt( n, a, y, mode )
call vmdcbrti(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdcbrt, vmdcbrt REAL, INTENT(IN) for vscbrt, vmscbrt DOUBLE PRECISION, INTENT(IN) for vdcbrt, vmdcbrt	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdcbrt, vmdcbrt REAL, INTENT(OUT) for vscbrt, vmscbrt DOUBLE PRECISION, INTENT(OUT) for vdcbrt, vmdcbrt	Array that specifies the output vector *y*.

Description

The v?Cbrt function computes a cube root of vector elements.

Special Values for Real Function v?Cbrt(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

v?InvCbrt

Computes an inverse cube root of vector elements.

Syntax

```
call vsinvcbrt( n, a, y )
call vsinvcbrti(n, a, inca, y, incy)
call vmsinvcbrt( n, a, y, mode )
call vmsinvcbrti(n, a, inca, y, incy, mode)
call vdinvcbrt( n, a, y )
call vdinvcbrti(n, a, inca, y, incy)
call vmdinvcbrt( n, a, y, mode )
call vmdinvcbrti(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdinvcbrt, vmdinvcbrt REAL, INTENT(IN) for vsinvcbrt, vmsinvcbrt DOUBLE PRECISION, INTENT(IN) for vdinvcbrt, vmdinvcbrt	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdinvcbrt, vmdinvcbrt REAL, INTENT(OUT) for vsinvcbrt, vmsinvcbrt DOUBLE PRECISION, INTENT(OUT) for vdinvcbrt, vmdinvcbrt	Array that specifies the output vector *y*.

Description

The v?InvCbrt function computes an inverse cube root of vector elements.

Special Values for Real Function v?InvCbrt(x)

Argument	Result	VM Error Status	Exception
+0	+∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
+∞	+0		
-∞	-0		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Pow2o3

Computes the cube root of the square of each vector element.

Syntax

```
call vspow2o3( n, a, y )
call vspow2o3i(n, a, inca, y, incy)
call vmspow2o3( n, a, y, mode )
call vmspow2o3i(n, a, inca, y, incy, mode)
call vdpow2o3( n, a, y )
call vdpow2o3i(n, a, inca, y, incy)
call vmdpow2o3( n, a, y, mode )
call vmdpow2o3i(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.

Name	Type	Description
a	DOUBLE PRECISION for vdpow2o3, vmdpow2o3 REAL, INTENT(IN) for vspow2o3, vmspow2o3 DOUBLE PRECISION, INTENT(IN) for vdpow2o3, vmdpow2o3	Arrays, specify the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdpow2o3, vmdpow2o3 REAL, INTENT(OUT) for vspow2o3, vmspow2o3 DOUBLE PRECISION, INTENT(OUT) for vdpow2o3, vmdpow2o3	Array, specifies the output vector *y*.

Description

The v?Pow2o3 function computes the cube root of the square of each vector element.

Special Values for Real Function v?Pow2o3(x)

Argument	Result	Exception
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

v?Pow3o2

Computes the square root of the cube of each vector element.

Syntax

```
call vspow3o2( n, a, y )
call vspow3o2i(n, a, inca, y, incy)
call vmspow3o2( n, a, y, mode )
call vmspow3o2i(n, a, inca, y, incy, mode)
call vdpow3o2( n, a, y )
```

```
call vdpow3o2i(n, a, inca, y, incy)
call vmdpow3o2( n, a, y, mode )
call vmdpow3o2i(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdpow3o2, vmdpow3o2 REAL, INTENT(IN) for vspow3o2, vmspow3o2 DOUBLE PRECISION, INTENT(IN) for vdpow3o2, vmdpow3o2	Arrays, specify the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Pow3o2 Function

Data Type	Threshold Limitations on Input Parameters
single precision	abs(*a*[i]) < (FLT_MAX)$^{2/3}$
double precision	abs(*a*[i]) < (DBL_MAX)$^{2/3}$

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdpow3o2, vmdpow3o2 REAL, INTENT(OUT) for vspow3o2, vmspow3o2 DOUBLE PRECISION, INTENT(OUT) for vdpow3o2, vmdpow3o2	Array, specifies the output vector *y*.

Description

The v?Pow3o2 function computes the square root of the cube of each vector element.

Special Values for Real Function v?Pow3o2(x)

Argument	Result	VM Error Status	Exception
X < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+0		
-0	-0		

Argument	Result	VM Error Status	Exception
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Pow

Computes a to the power b for elements of two vectors.

Syntax

```
call vspow( n, a, b, y )
call vspowi(n, a, inca, b, incb, y, incy)
call vmspow( n, a, b, y, mode )
call vmspowi(n, a, inca, b, incb, y, incy, mode)
call vdpow( n, a, b, y )
call vdpowi(n, a, inca, b, incb, y, incy)
call vmdpow( n, a, b, y, mode )
call vmdpowi(n, a, inca, b, incb, y, incy, mode)
call vcpow( n, a, b, y )
call vcpowi(n, a, inca, b, incb, y, incy)
call vmcpow( n, a, b, y, mode )
call vmcpowi(n, a, inca, b, incb, y, incy, mode)
call vzpow( n, a, b, y )
call vzpowi(n, a, inca, b, incb, y, incy)
call vmzpow( n, a, b, y, mode )
call vmzpowi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a, *b*	DOUBLE PRECISION for vdpow, vmdpow COMPLEX for vcpow, vmcpow DOUBLE COMPLEX for vzpow, vmzpow REAL, INTENT(IN) for vspow, vmspow	Arrays that specify the input vectors *a* and *b*.

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdpow, vmdpow COMPLEX, INTENT(IN) for vcpow, vmcpow DOUBLE COMPLEX, INTENT(IN) for vzpow, vmzpow	
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Pow Function

Data Type	Threshold Limitations on Input Parameters
single precision	abs(*a*[i]) < (FLT_MAX)$^{1/b[i]}$
double precision	abs(*a*[i]) < (DBL_MAX)$^{1/b[i]}$

Precision overflow thresholds for the complex v?Pow function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdpow, vmdpow COMPLEX for vcpow, vmcpow DOUBLE COMPLEX for vzpow, vmzpow REAL, INTENT(OUT) for vspow, vmspow DOUBLE PRECISION, INTENT(OUT) for vdpow, vmdpow COMPLEX, INTENT(OUT) for vcpow, vmcpow DOUBLE COMPLEX, INTENT(OUT) for vzpow, vmzpow	Array that specifies the output vector *y*.

Description

The v?Pow function computes *a* to the power *b* for elements of two vectors.

The real function v(s/d)Pow has certain limitations on the input range of *a* and *b* parameters. Specifically, if *a*[*i*] is positive, then *b*[*i*] may be arbitrary. For negative *a*[*i*], the value of *b*[*i*] must be an integer (either positive or negative).

The complex function v(c/z)Pow has no input range limitations.

Special values for Real Function v?Pow(x,y)

Argument 1 (X)	Argument 2 (Y)	Result	VM Error Status	Exception
+0	neg. odd integer	+∞	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. odd integer	-∞	VML_STATUS_ERRDOM	ZERODIVIDE
+0	neg. even integer	+∞	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. even integer	+∞	VML_STATUS_ERRDOM	ZERODIVIDE
+0	neg. non-integer	+∞	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. non-integer	+∞	VML_STATUS_ERRDOM	ZERODIVIDE
-0	pos. odd integer	+0		
-0	pos. odd integer	-0		
+0	pos. even integer	+0		
-0	pos. even integer	+0		
+0	pos. non-integer	+0		
-0	pos. non-integer	+0		
-1	+∞	+1		
-1	-∞	+1		
+1	any value	+1		
+1	+0	+1		
+1	-0	+1		
+1	+∞	+1		
+1	-∞	+1		
+1	QNAN	+1		
any value	+0	+1		
+0	+0	+1		
-0	+0	+1		
+∞	+0	+1		
-∞	+0	+1		
QNAN	+0	+1		
any value	-0	+1		
+0	-0	+1		
-0	-0	+1		
+∞	-0	+1		
-∞	-0	+1		
QNAN	-0	+1		
X < +0	non-integer	QNAN	VML_STATUS_ERRDOM	INVALID
|X| < 1	-∞	+∞		
+0	-∞	+∞	VML_STATUS_ERRDOM	ZERODIVIDE
-0	-∞	+∞	VML_STATUS_ERRDOM	ZERODIVIDE
|X| > 1	-∞	+0		
+∞	-∞	+0		
-∞	-∞	+0		
|X| < 1	+∞	+0		
+0	+∞	+0		
-0	+∞	+0		
|X| > 1	+∞	+∞		

Argument 1 (X)	Argument 2 (Y)	Result	VM Error Status	Exception
+∞	+∞	+∞		
-∞	+∞	+∞		
-∞	neg. odd integer	-0		
-∞	neg. even integer	+0		
-∞	neg. non-integer	+0		
-∞	pos. odd integer	-∞		
-∞	pos. even integer	+∞		
-∞	pos. non-integer	+∞		
+∞	X < +0	+0		
+∞	X > +0	+∞		
Big finite value*	Big finite value*	+/-∞	VML_STATUS_OVERFLOW	OVERFLOW
QNAN	QNAN	QNAN		
QNAN	SNAN	QNAN		INVALID
SNAN	QNAN	QNAN		INVALID
SNAN	SNAN	QNAN		INVALID

The complex double precision versions of this function, `vzPow` and `vmzPow`, are implemented in the EP accuracy mode only. If used in HA or LA mode, `vzPow` and `vmzPow` set the VM Error Status to `VML_STATUS_ACCURACYWARNING` (see the Values of the VM Status table).

* Overflow in a real function is supported only in the HA/LA accuracy modes. The overflow occurs when x and y are finite numbers, but the result is too large to fit the target precision. In this case, the function:

1. Returns ∞ in the result.
2. Raises the `OVERFLOW` exception.
3. Sets the VM Error Status to `VML_STATUS_OVERFLOW`.

Overflow in a complex function occurs (supported in the `HA`/`LA` accuracy modes only) when all `RE(x)`, `RE(y)`, `IM(x)`, `IM(y)` arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?Powx

Computes vector a to the scalar power b.

Syntax

```
call vspowx( n, a, b, y )
call vspowxi(n, a, inca, b, y, incy)
call vmspowx( n, a, b, y, mode )
call vmspowxi(n, a, inca, b, y, incy, mode)
call vdpowx( n, a, b, y )
call vdpowxi(n, a, inca, b, y, incy)
call vmdpowx( n, a, b, y, mode )
call vmdpowxi(n, a, inca, b, y, incy, mode)
call vcpowx( n, a, b, y )
```

```
call vcpowxi(n, a, inca, b, y, incy)
call vmcpowx( n, a, b, y, mode )
call vmcpowxi(n, a, inca, b, y, incy, mode)
call vzpowx( n, a, b, y )
call vzpowxi(n, a, inca, b, y, incy)
call vmzpowx( n, a, b, y, mode )
call vmzpowxi(n, a, inca, b, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Number of elements to be calculated.
a	DOUBLE PRECISION for vdpowx, vmdpowx COMPLEX for vcpowx, vmcpowx DOUBLE COMPLEX for vzpowx, vmzpowx REAL, INTENT(IN) for vspowx, vmspowx DOUBLE PRECISION, INTENT(IN) for vdpowx, vmdpowx COMPLEX, INTENT(IN) for vcpowx, vmcpowx DOUBLE COMPLEX, INTENT(IN) for vzpowx, vmzpowx	Array *a* that specifies the input vector
b	DOUBLE PRECISION for vdpowx, vmdpowx COMPLEX for vcpowx, vmcpowx DOUBLE COMPLEX for vzpowx, vmzpowx REAL, INTENT(IN) for vspowx, vmspowx DOUBLE PRECISION, INTENT(IN) for vdpowx, vmdpowx COMPLEX, INTENT(IN) for vcpowx, vmcpowx DOUBLE COMPLEX, INTENT(IN) for vzpowx, vmzpowx	Scalar value *b* that is the constant power.

Name	Type	Description
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Powx Function

Data Type	Threshold Limitations on Input Parameters
single precision	abs(*a*[i]) < (FLT_MAX)$^{1/b}$
double precision	abs(*a*[i]) < (DBL_MAX)$^{1/b}$

Precision overflow thresholds for the complex v?Powx function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdpowx, vmdpowx COMPLEX for vcpowx, vmcpowx DOUBLE COMPLEX for vzpowx, vmzpowx REAL, INTENT(OUT) for vspowx, vmspowx DOUBLE PRECISION, INTENT(OUT) for vdpowx, vmdpowx COMPLEX, INTENT(OUT) for vcpowx, vmcpowx DOUBLE COMPLEX, INTENT(OUT) for vzpowx, vmzpowx	Array that specifies the output vector *y*.

Description

The v?Powx function computes *a* to the power *b* for a vector *a* and a scalar *b*.

The real function v(s/d)Powx has certain limitations on the input range of *a* and *b* parameters. Specifically, if *a*[*i*] is positive, then *b* may be arbitrary. For negative *a*[*i*], the value of *b* must be an integer (either positive or negative).

The complex function v(c/z)Powx has no input range limitations.

Special values and VM Error Status treatment are the same as for the v?Pow function.

v?Powr

Computes a to the power b for elements of two vectors, where the elements of vector argument a are all non-negative.

Syntax

```
call vspowr (n, a, b, y )
call vspowri(n, a, inca, b, incb, y, incy)
```

```
call vmspowr (n, a, b, y, mode )
call vmspowri(n, a, inca, b, incb, y, incy, mode)
call vdpowr (n, a, b, y )
call vdpowri(n, a, inca, b, incb, y, incy)
call vmdpowr (n, a, b, y, mode )
call vmdpowri(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a, b	REAL for vsPowr REAL for vmsPowr DOUBLE PRECISION for vdPowr DOUBLE PRECISION for vmdPowr	Pointers to arrays containing the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vsPowr REAL for vmsPowr DOUBLE PRECISION for vdPowr DOUBLE PRECISION for vmdPowr	Pointer to an array containing the output vector *y*.

Description

The v?Powr function raises each element of vector *a* by the corresponding element of vector *b*. The elements of *a* are all nonnegative ($a_i \geq 0$).

Precision Overflow Thresholds for Real Function v?Powr

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < (\text{FLT\_MAX})^{1/b_i}$
double precision	$a_i < (\text{DBL\_MAX})^{1/b_i}$

Special values and VM Error Status treatment for v?Powr function are the same as for v?Pow, unless otherwise indicated in this table:

Special values for Real Function v?Powr(x)

Argument 1	Argument 2	Result	VM Error Status	Exception
$x < 0$	any value y	NAN	VML_STATUS_ERRDOM	INVALID
$0 < x < \infty$	±0	1		
±0	$-\infty < y < 0$	$+\infty$		
±0	$-\infty$	$+\infty$		
±0	$y > 0$	+0		
1	$-\infty < y < \infty$	1		
±0	±0	NAN		
$+\infty$	±0	NAN		
1	$+\infty$	NAN		
$x \geq 0$	NAN	NAN		
NAN	any value y	NAN		
$0 < x < 1$	$-\infty$	$+\infty$		
$x > 1$	$-\infty$	+0		
$0 \leq x < 1$	$+\infty$	+0		
$x > 1$	$+\infty$	$+\infty$		
$+\infty$	$x < +0$	+0		
$+\infty$	$x > +0$	$+\infty$		
QNAN	QNAN	QNAN	VML_STATUS_ERRDOM	
QNAN	SNAN	QNAN	VML_STATUS_ERRDOM	INVALID
SNAN	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
SNAN	SNAN	QNAN	VML_STATUS_ERRDOM	INVALID

See Also

Pow Computes *a* to the power *b* for elements of two vectors.

Powx Computes vector *a* to the scalar power *b*.

v?Hypot

Computes a square root of sum of two squared elements.

Syntax

```
call vshypot( n, a, b, y )
call vshypoti(n, a, inca, b, incb, y, incy)
call vmshypot( n, a, b, y, mode )
call vmshypoti(n, a, inca, b, incb, y, incy, mode)
call vdhypot( n, a, b, y )
call vdhypoti(n, a, inca, b, incb, y, incy)
call vmdhypot( n, a, b, y, mode )
call vmdhypoti(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	`INTEGER, INTENT(IN)`	Number of elements to be calculated.

a, *b*	`DOUBLE PRECISION` for `vdhypot`, `vmdhypot`
	`REAL, INTENT(IN)` for `vshypot`, `vmshypot`
	`DOUBLE PRECISION, INTENT(IN)` for `vdhypot, vmdhypot`
		Arrays that specify the input vectors *a* and *b*

inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	`INTEGER(KIND=8), INTENT(IN)`	Overrides global VM mode setting for this function call. See `vmlSetMode` for possible values and their description.

Precision Overflow Thresholds for Hypot Function

Data Type	Threshold Limitations on Input Parameters
single precision	`abs(a[i]) < sqrt(FLT_MAX)`
	`abs(b[i]) < sqrt(FLT_MAX)`
double precision	`abs(a[i]) < sqrt(DBL_MAX)`
	`abs(b[i]) < sqrt(DBL_MAX)`

Output Parameters

Name	Type	Description

y	`DOUBLE PRECISION` for `vdhypot`, `vmdhypot`
	`REAL, INTENT(OUT)` for `vshypot`, `vmshypot`
	`DOUBLE PRECISION, INTENT(OUT)` for `vdhypot, vmdhypot`
		Array that specifies the output vector *y*.

Description

The function `v?Hypot` computes a square root of sum of two squared elements.

Special values for Real Function v?Hypot(x)

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
-0	-0	+0	
+∞	any value	+∞	
any value	+∞	+∞	
SNAN	any value	QNAN	INVALID

Argument 1	Argument 2	Result	Exception
any value	SNAN	QNAN	INVALID
QNAN	any value	QNAN	
any value	QNAN	QNAN	

Exponential and Logarithmic Functions

v?Exp

Computes an exponential of vector elements.

Syntax

```
call vsexp( n, a, y )
call vsexpi(n, a, inca, y, incy)
call vmsexp( n, a, y, mode )
call vmsexpi(n, a, inca, y, incy, mode)
call vdexp( n, a, y )
call vdexpi(n, a, inca, y, incy)
call vmdexp( n, a, y, mode )
call vmdexpi(n, a, inca, y, incy, mode)
call vcexp( n, a, y )
call vcexpi(n, a, inca, y, incy)
call vmcexp( n, a, y, mode )
call vmcexpi(n, a, inca, y, incy, mode)
call vzexp( n, a, y )
call vzexpi(n, a, inca, y, incy)
call vmzexp( n, a, y, mode )
call vmzexpi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdexp, vmdexp COMPLEX for vcexp, vmcexp DOUBLE COMPLEX for vzexp, vmzexp REAL, INTENT(IN) for vsexp, vmsexp	Array, specifies the input vector *a*.

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdexp, vmdexp COMPLEX, INTENT(IN) for vcexp, vmcexp DOUBLE COMPLEX, INTENT(IN) for vzexp, vmzexp	
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Exp Function

Data Type	Threshold Limitations on Input Parameters
single precision	*a*[i] < Ln(FLT_MAX)
double precision	*a*[i] < Ln(DBL_MAX)

Precision overflow thresholds for the complex v?Exp function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdexp, vmdexp COMPLEX for vcexp, vmcexp DOUBLE COMPLEX for vzexp, vmzexp REAL, INTENT(OUT) for vsexp, vmsexp DOUBLE PRECISION, INTENT(OUT) for vdexp, vmdexp COMPLEX, INTENT(OUT) for vcexp, vmcexp DOUBLE COMPLEX, INTENT(OUT) for vzexp, vmzexp	Array, specifies the output vector *y*.

Description

The v?Exp function computes an exponential of vector elements.

Special Values for Real Function v?Exp(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
X < underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
+∞	+∞		

Argument	Result	VM Error Status	Exception
-∞	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Exp(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+0+i·0	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN INVALID	QNAN +i·QNAN INVALID
+i·Y	+0·CIS(Y)					+∞·CIS(Y)	QNAN +i·QNAN
+i·0	+0·CIS(0)		+1+i·0	+1+i·0		+∞+i·0	QNAN+i·0
-i·0	+0·CIS(0)		+1-i·0	+1-i·0		+∞-i·0	QNAN-i·0
-i·Y	+0·CIS(Y)					+∞·CIS(Y)	QNAN +i·QNAN
-i·∞	+0-i·0	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN INVALID	QNAN +i·QNAN INVALID
+i·NAN	+0+i·0	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN	QNAN +i·QNAN

Notes:

- raises the `INVALID` exception when real or imaginary part of the argument is `SNAN`
- raises the `INVALID` exception on argument `z=-∞+i·QNAN`
- raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when both `RE(z)` and `IM(z)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.

v?Exp2

Computes the base 2 exponential of vector elements.

Syntax

```
call vsexp2 (n, a, y)
call vsexp2i(n, a, inca, y, incy)
call vmsexp2 (n, a, y, mode)
call vmsexp2i(n, a, inca, y, incy, mode)
call vdexp2 (n, a, y)
call vdexp2i(n, a, inca, y, incy)
call vmdexp2 (n, a, y, mode)
call vmdexp2i(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vsExp2 REAL for vmsExp2 DOUBLE PRECISION for vdExp2 DOUBLE PRECISION for vmdExp2	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vsExp2 REAL for vmsExp2 DOUBLE PRECISION for vdExp2 DOUBLE PRECISION for vmdExp2	Pointer to an array containing the output vector *y*.

Description

The v?Exp2 function computes the base 2 exponential of vector elements.

Precision Overflow Thresholds for Real Function v?Exp2

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_2(\text{FLT\_MAX})$
double precision	$a_i < \log_2(\text{DBL\_MAX})$

See Special Value Notations for the conventions used in this table:

Special values for Real Function v?Exp2(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
x > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
x < underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
+∞	+∞		
-∞	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

Exp Computes an exponential of vector elements.
Exp10 Computes the base 10 exponential of vector elements.

v?Exp10

Computes the base 10 exponential of vector elements.

Syntax

```
call vsexp10 (n, a, y)
call vsexp10i(n, a, inca, y, incy)
call vmsexp10 (n, a, y, mode)
call vmsexp10i(n, a, inca, y, incy, mode)
call vdexp10 (n, a, y)
call vdexp10i(n, a, inca, y, incy)
call vmdexp10 (n, a, y, mode)
call vmdexp10i(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vsExp10 REAL for vmsExp10 DOUBLE PRECISION for vdExp10 DOUBLE PRECISION for vmdExp10	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vsExp10 REAL for vmsExp10 DOUBLE PRECISION for vdExp10 DOUBLE PRECISION for vmdExp10	Pointer to an array containing the output vector *y*.

Description

The `v?Exp10` function computes the base 10 exponential of vector elements.

Precision Overflow Thresholds for Real Function v?Exp10

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_{10}(\text{FLT\_MAX})$
double precision	$a_i < \log_{10}(\text{DBL\_MAX})$

See Special Value Notations for the conventions used in this table:

Special values for Real Function v?Pow(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
x > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
x < underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
+∞	+∞		
-∞	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

Exp Computes an exponential of vector elements.
Exp2 Computes the base 2 exponential of vector elements.

v?Expm1

Computes an exponential of vector elements decreased by 1.

Syntax

```
call vsexpm1( n, a, y )
call vsexpm1i(n, a, inca, y, incy)
call vmsexpm1( n, a, y, mode )
call vmsexpm1i(n, a, inca, y, incy, mode)
call vdexpm1( n, a, y )
call vdexpm1i(n, a, inca, y, incy)
call vdexpm1( n, a, y, mode )
call vmdexpm1i(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER, INTENT(IN)`	Specifies the number of elements to be calculated.

Name	Type	Description
a	DOUBLE PRECISION for vdexpm1, vmdexpm1 REAL, INTENT(IN) for vsexpm1, vmsexpm1 DOUBLE PRECISION, INTENT(IN) for vdexpm1, vmdexpm1	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Expm1 Function

Data Type	Threshold Limitations on Input Parameters
single precision	*a*[i] < Ln(FLT_MAX)
double precision	*a*[i] < Ln(DBL_MAX)

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdexpm1, vmdexpm1 REAL, INTENT(OUT) for vsexpm1, vmsexpm1 DOUBLE PRECISION, INTENT(OUT) for vdexpm1, vmdexpm1	Array that specifies the output vector *y*.

Description

The v?Expm1 function computes an exponential of vector elements decreased by 1.

Special Values for Real Function v?Expm1(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	+0		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
+∞	+∞		
-∞	-1		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Ln

Computes natural logarithm of vector elements.

Syntax

```
call vsln( n, a, y )
call vslni(n, a, inca, y, incy)
```

```
call vmsln( n, a, y, mode )
call vmslni(n, a, inca, y, incy, mode)
call vdln( n, a, y )
call vdlni(n, a, inca, y, incy)
call vmdln( n, a, y, mode )
call vmdlni(n, a, inca, y, incy, mode)
call vcln( n, a, y )
call vclni(n, a, inca, y, incy)
call vmcln( n, a, y, mode )
call vmclni(n, a, inca, y, incy, mode)
call vzln( n, a, y )
call vzlni(n, a, inca, y, incy)
call vmzln( n, a, y, mode )
call vmzlni(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdln, vmdln COMPLEX for vcln, vmcln DOUBLE COMPLEX for vzln, vmzln REAL, INTENT(IN) for vsln, vmsln DOUBLE PRECISION, INTENT(IN) for vdln, vmdln COMPLEX, INTENT(IN) for vcln, vmcln DOUBLE COMPLEX, INTENT(IN) for vzln, vmzln	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdln, vmdln COMPLEX for vcln, vmcln DOUBLE COMPLEX for vzln, vmzln REAL, INTENT(OUT) for vsln, vmsln DOUBLE PRECISION, INTENT(OUT) for vdln, vmdln COMPLEX, INTENT(OUT) for vcln, vmcln DOUBLE COMPLEX, INTENT(OUT) for vzln, vmzln	Array that specifies the output vector *y*.

Description

The v?Ln function computes natural logarithm of vector elements.

Special Values for Real Function v?Ln(x)

Argument	Result	VM Error Status	Exception
+1	+0		
X < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	-∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Ln(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i \cdot \frac{3\pi}{4}$	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	+∞+i·π					+∞+i·0	QNAN +i·QNAN INVALID
+i·0	+∞+i·π		-∞+i·π ZERODIVIDE	-∞+i·0 ZERODIVIDE		+∞+i·0	QNAN +i·QNAN INVALID

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
-i·0	+∞-i·π		-∞-i·π ZERODIVIDE	-∞-i·0 ZERODIVIDE		+∞-i·0	QNAN +i·QNAN INVALID
-i·Y	+∞-i·π					+∞-i·0	QNAN +i·QNAN INVALID
-i·∞	$+\infty - i \cdot \frac{3\pi}{4}$	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	+∞ +i·QNAN	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN	QNAN +i·QNAN INVALID

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`

v?Log2

Computes the base 2 logarithm of vector elements.

Syntax

```
call vslog2 (n, a, y)
call vslog2i(n, a, inca, y, incy)
call vmslog2 (n, a, y, mode)
call vmslog2i(n, a, inca, y, incy, mode)
call vdlog2 (n, a, y)
call vdlog2i(n, a, inca, y, incy)
call vmdlog2 (n, a, y, mode)
call vmdlog2i(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER`	Specifies the number of elements to be calculated.
a	`REAL` for `vslog2` `REAL` for `vmslog2`	Pointer to the array containing the input vector *a*.

Name	Type	Description
	DOUBLE PRECISION for vdlog2	
	DOUBLE PRECISION for vmdlog2	
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vslog2	Pointer to an array containing the output vector *y*.
	REAL for vmslog2	
	DOUBLE PRECISION for vdlog2	
	DOUBLE PRECISION for vmdlog2	

Description

The v?Log2 function computes the base 2 logarithm of vector elements.

See Special Value Notations for the conventions used in this table:

Special values for Real Function v?Log2(x)

Argument	Result	VM Error Status	Exception
+1	+0		
x < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	-∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

Ln Computes natural logarithm of vector elements.
Log10 Computes the base 10 logarithm of vector elements.

v?Log10

Computes the base 10 logarithm of vector elements.

Syntax

```
call vslog10( n, a, y )
call vslog10i(n, a, inca, y, incy)
call vmslog10( n, a, y, mode )
call vmslog10i(n, a, inca, y, incy, mode)
call vdlog10( n, a, y )
```

```
call vdlog10i(n, a, inca, y, incy)
call vmdlog10( n, a, y, mode )
call vmdlog10i(n, a, inca, y, incy, mode)
call vclog10( n, a, y )
call vclog10i(n, a, inca, y, incy)
call vmclog10( n, a, y, mode )
call vmclog10i(n, a, inca, y, incy, mode)
call vzlog10( n, a, y )
call vzlog10i(n, a, inca, y, incy)
call vmzlog10( n, a, y, mode )
call vmzlog10i(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdlog10, vmdlog10 COMPLEX for vclog10, vmclog10 DOUBLE COMPLEX for vzlog10, vmzlog10 REAL, INTENT(IN) for vslog10, vmslog10 DOUBLE PRECISION, INTENT(IN) for vdlog10, vmdlog10 COMPLEX, INTENT(IN) for vclog10, vmclog10 DOUBLE COMPLEX, INTENT(IN) for vzlog10, vmzlog10	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdlog10, vmdlog10 COMPLEX for vclog10, vmclog10 DOUBLE COMPLEX for vzlog10, vmzlog10 REAL, INTENT(OUT) for vslog10, vmslog10 DOUBLE PRECISION, INTENT(OUT) for vdlog10, vmdlog10 COMPLEX, INTENT(OUT) for vclog10, vmclog10 DOUBLE COMPLEX, INTENT(OUT) for vzlog10, vmzlog10	Array that specifies the output vector *y*.

Description

The v?Log10 function computes the base 10 logarithm of vector elements.

Special Values for Real Function v?Log10(x)

Argument	Result	VM Error Status	Exception
+1	+0		
X < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	-∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Log10(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·Y	$+\infty + i\frac{3}{4}\frac{\pi}{\ln(10)}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln(10)}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln(10)}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln(10)}$	$+\infty + i\frac{\pi}{2}$	$+\infty + i\frac{\pi}{4}\frac{1}{\ln(10)}$	+∞+i·QNAN INVALID
+i·Y	$+\infty + i\frac{\pi}{\ln(10)}$					+∞+i·0	QNAN +i·QNAN INVALID

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·0	$+\infty + i\frac{\pi}{\ln(10)}$		$-\infty + i\frac{\pi}{\ln(10)}$ ZERODIVIDE	-∞+i·0 ZERODIVIDE		+∞+i·0	QNAN +i·QNAN INVALID
-i·0	$+\infty - i\frac{\pi}{\ln(10)}$		$-\infty - i\frac{\pi}{\ln(10)}$ ZERODIVIDE	-∞-i·0 ZERODIVIDE		+∞-i·0	QNAN- i·QNAN INVALID
-i·Y	$+\infty - i\frac{\pi}{\ln(10)}$					+∞-i·0	QNAN +i·QNAN INVALID
-i	$+\infty + i\frac{3}{4}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln}$	$+\infty - i\frac{\pi}{2}$	$+\infty - i\frac{\pi}{4}\frac{1}{\ln(10)}$	+∞+i·QNAN
+i·NAN	+∞ +i·QNAN	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN	QNAN +i·QNAN INVALID

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`

v?Log1p

Computes a natural logarithm of vector elements that are increased by 1.

Syntax

```
call vslog1p( n, a, y )
call vslog1pi(n, a, inca, y, incy)
call vmslog1p( n, a, y, mode )
call vmslog1pi(n, a, inca, y, incy, mode)
call vdlog1p( n, a, y )
call vdlog1pi(n, a, inca, y, incy)
call vmdlog1p( n, a, y, mode )
call vmdlog1pi(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdlog1p, vmdlog1p REAL, INTENT(IN) for vslog1p, vmslog1p DOUBLE PRECISION, INTENT(IN) for vdlog1p, vmdlog1p	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdlog1p, vmdlog1p REAL, INTENT(OUT) for vslog1p, vmslog1p DOUBLE PRECISION, INTENT(OUT) for vdlog1p, vmdlog1p	Array that specifies the output vector *y*.

Description

The v?Log1p function computes a natural logarithm of vector elements that are increased by 1.

Special Values for Real Function v?Log1p(x)

Argument	Result	VM Error Status	Exception
-1	-∞	VML_STATUS_SING	ZERODIVIDE
X < -1	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+0		
-0	-0		
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Logb

Computes the exponents of the elements of input vector a.

Syntax

```
call vslogb (n, a, y)
```

```
call vslogbi(n, a, inca, y, incy)
call vmslogb (n, a, y, mode)
call vmslogbi(n, a, inca, y, incy, mode)
call vdlogb (n, a, y)
call vdlogbi(n, a, inca, y, incy)
call vmdlogb (n, a, y, mode)
call vmdlogbi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vslogb REAL for vmslogb DOUBLE PRECISION for vdlogb DOUBLE PRECISION for vmdlogb	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vslogb REAL for vmslogb DOUBLE PRECISION for vdlogb DOUBLE PRECISION for vmdlogb	Pointer to an array containing the output vector *y*.

Description

The v?Logb function computes the exponents of the elements of the input vector *a*. For each element a_i of vector *a*, this is the integral part of $\log_2|a_i|$. The returned value is exact and is independent of the current rounding direction mode.

See Special Value Notations for the conventions used in this table:

Special values for Real Function v?Logb(x)

Argument	Result	VM Error Status	Exception
+0	-∞	VML_STATUS_ERRDOM	ZERODIVIDE
-0	-∞	VML_STATUS_ERRDOM	ZERODIVIDE

Argument	Result	VM Error Status	Exception
-∞	+∞		
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

Trigonometric Functions

v?Cos

Computes cosine of vector elements.

Syntax

```
call vscos( n, a, y )
call vscosi(n, a, inca, y, incy)
call vmscos( n, a, y, mode )
call vmscosi(n, a, inca, y, incy, mode)
call vdcos( n, a, y )
call vdcosi(n, a, inca, y, incy)
call vmdcos( n, a, y, mode )
call vmdcosi(n, a, inca, y, incy, mode)
call vccos( n, a, y )
call vccosi(n, a, inca, y, incy)
call vmccos( n, a, y, mode )
call vmccosi(n, a, inca, y, incy, mode)
call vzcos( n, a, y )
call vzcosi(n, a, inca, y, incy)
call vmzcos( n, a, y, mode )
call vmzcosi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdcos, vmdcos COMPLEX for vccos, vmccos	Array that specifies the input vector *a*.

Name	Type	Description
	DOUBLE PRECISION for vzcos, vmzcos REAL, INTENT(IN) for vscos, vmscos DOUBLE PRECISION, INTENT(IN) for vdcos, vmdcos COMPLEX, INTENT(IN) for vccos, vmccos DOUBLE PRECISION, INTENT(IN) for vzcos, vmzcos	
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdcos, vmdcos COMPLEX for vccos, vmccos DOUBLE PRECISION for vzcos, vmzcos REAL, INTENT(OUT) for vscos, vmscos DOUBLE PRECISION, INTENT(OUT) for vdcos, vmdcos COMPLEX, INTENT(OUT) for vccos, vmccos DOUBLE PRECISION, INTENT(OUT) for vzcos, vmzcos	Array that specifies the output vector *y*.

Description

The v?Cos function computes cosine of vector elements.

Note that arguments abs(*a*[i]) ≤ 2^{13} and abs(*a*[i]) ≤ 2^{16} for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Cos(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		

Argument	Result	VM Error Status	Exception
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

`Cos(z) = Cosh(i*z).`

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

v?Sin

Computes sine of vector elements.

Syntax

```
call vssin( n, a, y )
call vssini(n, a, inca, y, incy)
call vmssin( n, a, y, mode )
call vmssini(n, a, inca, y, incy, mode)
call vdsin( n, a, y )
call vdsini(n, a, inca, y, incy)
call vmdsin( n, a, y, mode )
call vmdsini(n, a, inca, y, incy, mode)
call vcsin( n, a, y )
call vcsini(n, a, inca, y, incy)
call vmcsin( n, a, y, mode )
call vmcsini(n, a, inca, y, incy, mode)
call vzsin( n, a, y )
call vzsini(n, a, inca, y, incy)
call vmzsin( n, a, y, mode )
call vmzsini(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER, INTENT(IN)`	Specifies the number of elements to be calculated.

Name	Type	Description
a	DOUBLE PRECISION for vdsin, vmdsin COMPLEX for vcsin, vmcsin DOUBLE PRECISION for vzsin, vmzsin REAL, INTENT(IN) for vssin, vmssin DOUBLE PRECISION, INTENT(IN) for vdsin, vmdsin COMPLEX, INTENT(IN) for vcsin, vmcsin DOUBLE PRECISION, INTENT(IN) for vzsin, vmzsin	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdsin, vmdsin COMPLEX for vcsin, vmcsin DOUBLE PRECISION for vzsin, vmzsin REAL, INTENT(OUT) for vssin, vmssin DOUBLE PRECISION, INTENT(OUT) for vdsin, vmdsin COMPLEX, INTENT(OUT) for vcsin, vmcsin DOUBLE PRECISION, INTENT(OUT) for vzsin, vmzsin	Array that specifies the output vector *y*.

Description

The function computes sine of vector elements.

Note that arguments abs(*a*[i]) $\leq 2^{13}$ and abs(*a*[i]) $\leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Sin(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

`Sin(z) = -i*Sinh(i*z).`

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

v?SinCos

Computes sine and cosine of vector elements.

Syntax

```
call vssincos( n, a, y, z )
call vssincosi(n, a, inca, y, incy, z, incz)
call vmssincos( n, a, y, z, mode )
call vmssincosi(n, a, inca, y, incy, z, incz, mode)
call vdsincos( n, a, y, z )
call vdsincosi(n, a, inca, y, incy, z, incz)
call vmdsincos( n, a, y, z, mode )
call vmdsincosi(n, a, inca, y, incy, z, incz, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER, INTENT(IN)`	Specifies the number of elements to be calculated.

a	`DOUBLE PRECISION` for `vdsincos, vmdsincos`
	`REAL, INTENT(IN)` for `vssincos, vmssincos`
	`DOUBLE PRECISION, INTENT(IN)` for `vdsincos, vmdsincos`
		Array that specifies the input vector *a*.

Name	Type	Description
inca, *incy*, *incz*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *y*, and *z*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y, *z*	DOUBLE PRECISION for vdsincos, vmdsincos REAL, INTENT(OUT) for vssincos, vmssincos DOUBLE PRECISION, INTENT(OUT) for vdsincos, vmdsincos	Arrays that specify the output vectors *y* (for sine values) and *z* (for cosine values).

Description

The function computes sine and cosine of vector elements.

Note that arguments abs(*a*[i]) ≤ 2^{13} and abs(*a*[i]) ≤ 2^{16} for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?SinCos(x)

Argument	Result 1	Result 2	VM Error Status	Exception
+0	+0	+1		
-0	-0	+1		
+∞	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN	QNAN		
SNAN	QNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

Sin(z) = –i*Sinh(i*z).

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

v?CIS

Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Syntax

```
call vccis( n, a, y )
call vccisi(n, a, inca, y, incy)
call vmccis( n, a, y, mode )
call vmccisi(n, a, inca, y, incy, mode)
call vzcis( n, a, y )
call vzcisi(n, a, inca, y, incy)
call vmzcis( n, a, y, mode )
call vmzcisi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vzcis, vmzcis REAL, INTENT(IN) for vccis, vmccis DOUBLE PRECISION, INTENT(IN) for vzcis, vmzcis	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE COMPLEX for vzcis, vmzcis COMPLEX, INTENT(OUT) for vccis, vmccis DOUBLE COMPLEX, INTENT(OUT) for vzcis, vmzcis	Array that specifies the output vector *y*.

Description

The v?CIS function computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?CIS(x)

x	CIS(x)
+ ∞	QNAN+i·QNAN INVALID
+ 0	+1+i·0
- 0	+1-i·0
- ∞	QNAN+i·QNAN INVALID
NAN	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when the argument is `SNAN`
- raises `INVALID` exception and sets the VM Error Status to `VML_STATUS_ERRDOM` for `x=+∞`, `x=-∞`

v?Tan

Computes tangent of vector elements.

Syntax

```
call vstan( n, a, y )
call vstani(n, a, inca, y, incy)
call vmstan( n, a, y, mode )
call vmstani(n, a, inca, y, incy, mode)
call vdtan( n, a, y )
call vdtani(n, a, inca, y, incy)
call vmdtan( n, a, y, mode )
call vmdtani(n, a, inca, y, incy, mode)
call vctan( n, a, y )
call vctani(n, a, inca, y, incy)
call vmctan( n, a, y, mode )
call vmctani(n, a, inca, y, incy, mode)
call vztan( n, a, y )
call vztani(n, a, inca, y, incy)
call vmztan( n, a, y, mode )
call vmztani(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdtan, vmdtan COMPLEX for vctan, vmctan DOUBLE COMPLEX for vztan, vmztan REAL, INTENT(IN) for vstan, vmstan DOUBLE PRECISION, INTENT(IN) for vdtan, vmdtan COMPLEX, INTENT(IN) for vctan, vmctan DOUBLE COMPLEX, INTENT(IN) for vztan, vmztan	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdtan, vmdtan COMPLEX for vctan, vmctan DOUBLE COMPLEX for vztan, vmztan REAL, INTENT(OUT) for vstan, vmstan DOUBLE PRECISION, INTENT(OUT) for vdtan, vmdtan COMPLEX, INTENT(OUT) for vctan, vmctan DOUBLE COMPLEX, INTENT(OUT) for vztan, vmztan	Array that specifies the output vector *y*.

Description

The v?Tan function computes tangent of vector elements.

Note that arguments abs(a[i]) ≤ 2^{13} and abs(a[i]) ≤ 2^{16} for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Tan(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula Tan(z) = –i*Tanh(i*z).

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

v?Acos

Computes inverse cosine of vector elements.

Syntax

```
call vsacos( n, a, y )
call vsacosi(n, a, inca, y, incy)
call vmsacos( n, a, y, mode )
call vmsacosi(n, a, inca, y, incy, mode)
call vdacos( n, a, y )
call vdacosi(n, a, inca, y, incy)
call vmdacos( n, a, y, mode )
call vmdacosi(n, a, inca, y, incy, mode)
call vcacos( n, a, y )
call vcacosi(n, a, inca, y, incy)
call vmcacos( n, a, y, mode )
call vmcacosi(n, a, inca, y, incy, mode)
call vzacos( n, a, y )
call vzacosi(n, a, inca, y, incy)
call vmzacos( n, a, y, mode )
call vmzacosi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdacos, vmdacos COMPLEX for vcacos, vmcacos DOUBLE COMPLEX for vzacos, vmzacos REAL, INTENT(IN) for vsacos, vmsacos DOUBLE PRECISION, INTENT(IN) for vdacos, vmdacos COMPLEX, INTENT(IN) for vcacos, vmcacos DOUBLE COMPLEX, INTENT(IN) for vzacos, vmzacos	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdacos, vmdacos COMPLEX for vcacos, vmcacos DOUBLE COMPLEX for vzacos, vmzacos REAL, INTENT(OUT) for vsacos, vmsacos DOUBLE PRECISION, INTENT(OUT) for vdacos, vmdacos COMPLEX, INTENT(OUT) for vcacos, vmcacos DOUBLE COMPLEX, INTENT(OUT) for vzacos, vmzacos	Array that specifies the output vector *y*.

Description

The `v?Acos` function computes inverse cosine of vector elements.

Special Values for Real Function v?Acos(x)

Argument	Result	VM Error Status	Exception
+0	+π/2		
-0	+π/2		
+1	+0		
-1	+π		
|X| > 1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Acos(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+3·π/4-i·∞	+π/2-i·∞	+π/2-i·∞	+π/2-i·∞	+π/2-i·∞	+π/4-i·∞	QNAN-i·∞
+i·Y	+π-i·∞					+0-i·∞	QNAN +i·QNAN
+i·0	+π-i·∞		+π/2-i·0	+π/2-i·0		+0-i·∞	QNAN +i·QNAN
-i·0	+π+i·∞		+π/2+i·0	+π/2+i·0		+0+i·∞	QNAN +i·QNAN
-i·Y	+π+i·∞					+0+i·∞	QNAN +i·QNAN
-i·∞	+3π/4+i·∞	+π/2+i·∞	+π/2+i·∞	+π/2+i·∞	+π/2+i·∞	+π/4+i·∞	QNAN+i·∞
+i·NAN	QNAN+i·∞	QNAN +i·QNAN	+π/ 2+i·QNAN	+π/ 2+i·QNAN	QNAN +i·QNAN	QNAN+i·∞	QNAN +i·QNAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- `Acos(CONJ(z))=CONJ(Acos(z))`.

v?Asin

Computes inverse sine of vector elements.

Syntax

```
call vsasin( n, a, y )
call vsasini(n, a, inca, y, incy)
call vmsasin( n, a, y, mode )
call vmsasini(n, a, inca, y, incy, mode)
call vdasin( n, a, y )
call vdasini(n, a, inca, y, incy)
```

```
call vmdasin( n, a, y, mode )
call vmdasini(n, a, inca, y, incy, mode)
call vcasin( n, a, y )
call vcasini(n, a, inca, y, incy)
call vmcasin( n, a, y, mode )
call vmcasini(n, a, inca, y, incy, mode)
call vzasin( n, a, y )
call vzasini(n, a, inca, y, incy)
call vmzasin( n, a, y, mode )
call vmzasini(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdasin, vmdasin COMPLEX for vcasin, vmcasin DOUBLE COMPLEX for vzasin, vmzasin REAL, INTENT(IN) for vsasin, vmsasin DOUBLE PRECISION, INTENT(IN) for vdasin, vmdasin COMPLEX, INTENT(IN) for vcasin, vmcasin DOUBLE COMPLEX, INTENT(IN) for vzasin, vmzasin	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdasin, vmdasin	Array that specifies the output vector *y*.

Name	Type	Description
	`COMPLEX` for `vcasin`, `vmcasin`	
	`DOUBLE COMPLEX` for `vzasin`, `vmzasin`	
	`REAL`, `INTENT(OUT)` for `vsasin`, `vmsasin`	
	`DOUBLE PRECISION`, `INTENT(OUT)` for `vdasin`, `vmdasin`	
	`COMPLEX`, `INTENT(OUT)` for `vcasin`, `vmcasin`	
	`DOUBLE COMPLEX`, `INTENT(OUT)` for `vzasin`, `vmzasin`	

Description

The `v?Asin` function computes inverse sine of vector elements.

Special Values for Real Function v?Asin(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+1	+π/2		
-1	-π/2		
|X| > 1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

`Asin(z) = -i*Asinh(i*z).`

v?Atan

Computes inverse tangent of vector elements.

Syntax

```
call vsatan( n, a, y )
call vsatani(n, a, inca, y, incy)
call vmsatan( n, a, y, mode )
call vmsatani(n, a, inca, y, incy, mode)
call vdatan( n, a, y )
call vdatani(n, a, inca, y, incy)
call vmdatan( n, a, y, mode )
call vmdatani(n, a, inca, y, incy, mode)
call vcatan( n, a, y )
```

```
call vcatani(n, a, inca, y, incy)
call vmcatan( n, a, y, mode )
call vmcatani(n, a, inca, y, incy, mode)
call vzatan( n, a, y )
call vzatani(n, a, inca, y, incy)
call vmzatan( n, a, y, mode )
call vmzatani(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdatan, vmdatan COMPLEX for vcatan, vmcatan DOUBLE COMPLEX for vzatan, vmzatan REAL, INTENT(IN) for vsatan, vmsatan DOUBLE PRECISION, INTENT(IN) for vdatan, vmdatan COMPLEX, INTENT(IN) for vcatan, vmcatan DOUBLE COMPLEX, INTENT(IN) for vzatan, vmzatan	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdatan, vmdatan COMPLEX for vcatan, vmcatan DOUBLE COMPLEX for vzatan, vmzatan	Array that specifies the output vector *y*.

Name	Type	Description
	`REAL, INTENT(OUT)` for `vsatan, vmsatan`	
	`DOUBLE PRECISION, INTENT(OUT)` for `vdatan, vmdatan`	
	`COMPLEX, INTENT(OUT)` for `vcatan, vmcatan`	
	`DOUBLE COMPLEX, INTENT(OUT)` for `vzatan, vmzatan`	

Description

The `v?Atan` function computes inverse tangent of vector elements.

Special Values for Real Function v?Atan(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+∞	+π/2		
-∞	-π/2		
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

`Atan(z) = -i*Atanh(i*z).`

v?Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

```
call vsatan2( n, a, b, y )
call vsatan2i(n, a, inca, b, incb, y, incy)
call vmsatan2( n, a, b, y, mode )
call vmsatan2i(n, a, inca, b, incb, y, incy, mode)
call vdatan2( n, a, b, y )
call vdatan2i(n, a, inca, b, incb, y, incy)
call vmdatan2( n, a, b, y, mode )
call vmdatan2i(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a, *b*	DOUBLE PRECISION for vdatan2, vmdatan2 REAL, INTENT(IN) for vsatan2, vmsatan2 DOUBLE PRECISION, INTENT(IN) for vdatan2, vmdatan2	Arrays that specify the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdatan2, vmdatan2 REAL, INTENT(OUT) for vsatan2, vmsatan2 DOUBLE PRECISION, INTENT(OUT) for vdatan2, vmdatan2	Array that specifies the output vector *y*.

Description

The v?Atan2 function computes four-quadrant inverse tangent of elements of two vectors.

The elements of the output vector*y* are computed as the four-quadrant arctangent of *a[i] / b[i]*.

Special values for Real Function v?Atan2(x)

Argument 1	Argument 2	Result	Exception
-∞	-∞	-3*π/4	
-∞	X < +0	-π/2	
-∞	-0	-π/2	
-∞	+0	-π/2	
-∞	X > +0	-π/2	
-∞	+∞	-π/4	
X < +0	-∞	-π	
X < +0	-0	-π/2	
X < +0	+0	-π/2	
X < +0	+∞	-0	
-0	-∞	-π	
-0	X < +0	-π	

Argument 1	Argument 2	Result	Exception
-0	-0	-π	
-0	+0	-0	
-0	X > +0	-0	
-0	+∞	-0	
+0	-∞	+π	
+0	X < +0	+π	
+0	-0	+π	
+0	+0	+0	
+0	X > +0	+0	
+0	+∞	+0	
X > +0	-∞	+π	
X > +0	-0	+π/2	
X > +0	+0	+π/2	
X > +0	+∞	+0	
+∞	-∞	+3*π/4	
+∞	X < +0	+π/2	
+∞	-0	+π/2	
+∞	+0	+π/2	
+∞	X > +0	+π/2	
+∞	+∞	+π/4	
X > +0	QNAN	QNAN	
X > +0	SNAN	QNAN	INVALID
QNAN	X > +0	QNAN	
SNAN	X > +0	QNAN	INVALID
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	INVALID
SNAN	QNAN	QNAN	INVALID
SNAN	SNAN	QNAN	INVALID

v?Cospi

Computes the cosine of vector elements multiplied by π.

Syntax

```
call vscospi (n, a, y)
call vscospii(n, a, inca, y, incy)
call vmscospi (n, a, y, mode)
call vmscospii(n, a, inca, y, incy, mode)
call vdcospi (n, a, y)
call vdcospii(n, a, inca, y, incy)
call vmdcospi (n, a, y, mode)
call vmdcospii(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vscospi REAL for vmscospi DOUBLE PRECISION for vdcospi DOUBLE PRECISION for vmdcospi	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vscospi REAL for vmscospi DOUBLE PRECISION for vdcospi DOUBLE PRECISION for vmdcospi	Pointer to an array containing the output vector *y*.

Description

The v?Cospi function computes the cosine of vector elements multiplied by *π*. For an argument *x*, the function computes cos(*π***x*).

Special values for Real Function v?Cospi(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
n + 0.5, for any integer *n* where *n* + 0.5 is representable	+0		
±∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Application Notes

If arguments abs(a_i) ≤ 2^{22} for single precision or abs(a_i) ≤ 2^{43} for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

Cos Computes cosine of vector elements.

Cosd Computes the cosine of vector elements multiplied by *π*/180.

v?Sinpi

Computes the sine of vector elements multiplied by π.

Syntax

```
call vssinpi (n, a, y)
call vssinpii(n, a, inca, y, incy)
call vmssinpi (n, a, y, mode)
call vmssinpii(n, a, inca, y, incy, mode)
call vdsinpi (n, a, y)
call vdsinpii(n, a, inca, y, incy)
call vmdsinpi (n, a, y, mode)
call vmdsinpii(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vssinpi REAL for vmssinpi DOUBLE PRECISION for vdsinpi DOUBLE PRECISION for vmdsinpi	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description

y	`REAL` for `vssinpi`
	`REAL` for `vmssinpi`
	`DOUBLE PRECISION` for `vdsinpi`
	`DOUBLE PRECISION` for `vmdsinpi`
		Pointer to an array containing the output vector *y*.

Description

The `v?Sinpi` function computes the sine of vector elements multiplied by *π*. For an argument *x*, the function computes sin(*π***x*).

Special values for Real Function v?Sinpi(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+*n*, positive integer	+0		
-*n*, negative integer	-0		
±∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Application Notes

If arguments abs(a_i) ≤ 2^{22} for single precision or abs(a_i) ≤ 2^{51} for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

Sin Computes sine of vector elements.

Sind Computes the sine of vector elements multiplied by *π*/180.

v?Tanpi

Computes the tangent of vector elements multiplied by π.

Syntax

```
call vstanpi (n, a, y)
call vstanpii(n, a, inca, y, incy)
call vmstanpi (n, a, y, mode)
call vmstanpii(n, a, inca, y, incy, mode)
call vdtanpi (n, a, y)
call vdtanpii(n, a, inca, y, incy)
call vmdtanpi (n, a, y, mode)
call vmdtanpii(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vstanpi REAL for vmstanpi DOUBLE PRECISION for vdtanpi DOUBLE PRECISION for vmdtanpi	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vstanpi REAL for vmstanpi DOUBLE PRECISION for vdtanpi DOUBLE PRECISION for vmdtanpi	Pointer to an array containing the output vector *y*.

Description

The v?Tanpi function computes the tangent of vector elements multiplied by π. For an argument x, the function computes $\tan(\pi * x)$.

Special values for Real Function v?Tanpi(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
±∞	QNAN	VML_STATUS_ERRDOM	INVALID
n, even integer	copysign(0.0, *n*)		
n, odd integer	copysign(0.0, -*n*)		
n + 0.5, for *n* even integer and *n* + 0.5 representable	+∞		
n + 0.5, for *n* odd integer and *n* + 0.5 representable	-∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

The copysign(x, y) function returns the first vector argument x with the sign changed to match that of the second argument y.

Application Notes

If arguments abs(a_i) ≤ 2^{13} for single precision or abs(a_i) ≤ 2^{67} for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

Tan Computes tangent of vector elements.

Tand Computes the tangent of vector elements multiplied by *π*/180.

v?Acospi

Computes the inverse cosine of vector elements divided by π.

Syntax

```
call vsacospi (n, a, y)
call vsacospii(n, a, inca, y, incy)
call vmsacospi (n, a, y, mode)
call vmsacospii(n, a, inca, y, incy, mode)
call vdacospi (n, a, y)
call vdacospii(n, a, inca, y, incy)
call vmdacospi (n, a, y, mode)
call vmdacospii(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vsacospi REAL for vmsacospi DOUBLE PRECISION for vdacospi DOUBLE PRECISION for vmdacospi	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description

y	`REAL` for `vsacospi`
	`REAL` for `vmsacospi`
	`DOUBLE PRECISION` for `vdacospi`
	`DOUBLE PRECISION` for `vmdacospi`
		Pointer to an array containing the output vector *y*.

Description

The `v?Acospi` function computes the inverse cosine of vector elements divided by *π*. For an argument *x*, the function computes acos(*x*)/*π*.

See Special Value Notations for the conventions used in this table:

Special values for Real Function v?Acospi(x)

Argument	Result	VM Error Status	Exception
+0	+1/2		
-0	+1/2		
+1	+0		
-1	+1		
|*x*| > 1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
- ∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

Acos Computes inverse cosine of vector elements.

v?Asinpi

Computes the inverse sine of vector elements divided by π.

Syntax

```
call vsasinpi (n, a, y)
call vsasinpii(n, a, inca, y, incy)
call vmsasinpi (n, a, y, mode)
call vmsasinpii(n, a, inca, y, incy, mode)
call vdasinpi (n, a, y)
call vdasinpii(n, a, inca, y, incy)
call vmdasinpi (n, a, y, mode)
call vmdasinpii(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER`	Specifies the number of elements to be calculated.

a	`REAL` for `vsasinpi`
	`REAL` for `vmsasinpi`
	`DOUBLE PRECISION` for `vdasinpi`
	`DOUBLE PRECISION` for `vmdasinpi`
		Pointer to the array containing the input vector *a*.

inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	`INTEGER (KIND=8)`	Overrides the global VM *mode* setting for this function call. See `vmlSetMode` for possible values and their description.

Output Parameters

Name	Type	Description

y	`REAL` for `vsasinpi`
	`REAL` for `vmsasinpi`
	`DOUBLE PRECISION` for `vdasinpi`
	`DOUBLE PRECISION` for `vmdasinpi`
		Pointer to an array containing the output vector *y*.

Description

The `v?Asinpi` function computes the inverse sine of vector elements divided by *π*. For an argument *x*, the function computes asin(*x*)/*π*.

Special values for Real Function v?Asinpi(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+1	+1/2		
-1	-1/2		
|*x*| > 1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

Asin Computes inverse sine of vector elements.

v?Atanpi

Computes the inverse tangent of vector elements divided by π.

Syntax

```
call vsatanpi (n, a, y)
call vsatanpii(n, a, inca, y, incy)
call vmsatanpi (n, a, y, mode)
call vmsatanpii(n, a, inca, y, incy, mode)
call vdatanpi (n, a, y)
call vdatanpii(n, a, inca, y, incy)
call vmdatanpi (n, a, y, mode)
call vmdatanpii(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vsatanpi REAL for vmsatanpi DOUBLE PRECISION for vdatanpi DOUBLE PRECISION for vmdatanpi	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vsatanpi REAL for vmsatanpi DOUBLE PRECISION for vdatanpi DOUBLE PRECISION for vmdatanpi	Pointer to an array containing the output vector *y*.

Description

The v?Atanpi function computes the inverse tangent of vector elements divided by *π*. For an argument *x*, the function computes atan(*x*)/*π*.

Special values for Real Function v?Atanpi(x)

Argument	Result	VM Error Status	Exception
+0	+0		

Argument	Result	VM Error Status	Exception
-0	-0		
+∞	+1/2		
-∞	-1/2		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

Atan Computes inverse tangent of vector elements.

v?Atan2pi

Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by π.

Syntax

```
call vsatan2pi (n, a, b, y)
call vsatan2pii(n, a, inca, b, incb, y, incy)
call vmsatan2pi (n, a, b, y, mode)
call vmsatan2pii(n, a, inca, b, incb, y, incy, mode)
call vdatan2pi (n, a, b, y)
call vdatan2pii(n, a, inca, b, incb, y, incy)
call vmdatan2pi (n, a, b, y, mode)
call vmdatan2pii(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a, *b*	REAL for vsatan2pi REAL for vmsatan2pi DOUBLE PRECISION for vdatan2pi DOUBLE PRECISION for vmdatan2pi	Pointers to the arrays containing the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description

y	`REAL` for `vsatan2pi`
	`REAL` for `vmsatan2pi`
	`DOUBLE PRECISION` for `vdatan2pi`
	`DOUBLE PRECISION` for `vmdatan2pi`
		Pointer to an array containing the output vector *y*.

Description

The `v?Atan2pi` function computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by π.

For the elements of the output vector *y*, the function computers the four-quadrant arctangent of a_i/b_i, with the result divided by π.

Special values for Real Function v?Atan2pi(x, y)

Argument 1	Argument 2	Result	Exception
-∞	-∞	-3/4	
-∞	*x* < +0	-1/2	
-∞	-0	+1/2	
-∞	+0	-1/2	
-∞	*x* > +0	-1/2	
-∞	+∞	-1/4	
x < +0	-∞	-1	
x < +0	-0	-1/2	
x < +0	+0	-1/2	
x < +0	+∞	-0	
-0	-∞	-1	
-0	*x* < +0	-1	
-0	-0	-1	
-0	+0	-0	
-0	*x* > +0	-0	
-0	+∞	-0	
+0	-∞	+1	
+0	*x* < +0	+1	
+0	-0	+1	
+0	+0	+0	
+0	*x* > +0	+0	
+0	+∞	+0	
x > +0	-∞	+1	
x > +0	-0	+1/2	
x > +0	+0	+1/2	
x > +0	+∞	+1/4	
+∞	-∞	+3/4	
+∞	*x* < +0	+1/2	
+∞	-0	+1/2	
+∞	+0	+1/2	

Argument 1	Argument 2	Result	Exception
+∞	*x* > +0	+1/2	
+∞	+∞	+1/4	
x > +0	QNAN	QNAN	
x > +0	SNAN	QNAN	INVALID
QNAN	*x* > +0	QNAN	
SNAN	*x* > +0	QNAN	INVALID
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	INVALID
SNAN	QNAN	QNAN	INVALID
SNAN	SNAN	QNAN	INVALID

See Also

Atan2 Computes four-quadrant inverse tangent of elements of two vectors.

v?Cosd

Computes the cosine of vector elements multiplied by π/180.

Syntax

```
call vscosd (n, a, y)
call vscosdi(n, a, inca, y, incy)
call vmscosd (n, a, y, mode)
call vmscosdi(n, a, inca, y, incy, mode)
call vdcosd (n, a, y)
call vdcosdi(n, a, inca, y, incy)
call vmdcosd (n, a, y, mode)
call vmdcosdi(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER`	Specifies the number of elements to be calculated.

a	`REAL` for `vscosd`
	`REAL` for `vmscosd`
	`DOUBLE PRECISION` for `vdcosd`
	`DOUBLE PRECISION` for `vmdcosd`
		Pointer to the array containing the input vector *a*.

inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.

Name	Type	Description
mode	`INTEGER (KIND=8)`	Overrides the global VM *mode* setting for this function call. See `vmlSetMode` for possible values and their description.

Output Parameters

Name	Type	Description

y	`REAL` for `vscosd`
	`REAL` for `vmscosd`
	`DOUBLE PRECISION` for `vdcosd`
	`DOUBLE PRECISION` for `vmdcosd`
		Pointer to an array containing the output vector *y*.

Description

The `v?Cosd` function computes the cosine of vector elements multiplied by $\pi/180$. For an argument x, the function computes $\cos(\pi * x/180)$.

Special values for Real Function v?Cosd(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
±∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Application Notes

If arguments $\text{abs}(a_i) \le 2^{24}$ for single precision or $\text{abs}(a_i) \le 2^{52}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

Cos Computes cosine of vector elements.

Cospi Computes the cosine of vector elements multiplied by π.

v?Sind

Computes the sine of vector elements multiplied by π/180.

Syntax

```
call vssind (n, a, y)
call vssindi(n, a, inca, y, incy)
call vmssind (n, a, y, mode)
call vmssindi(n, a, inca, y, incy, mode)
call vdsind (n, a, y)
call vdsindi(n, a, inca, y, incy)
```

```
call vmdsind (n, a, y, mode)
call vmdsindi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vssind REAL for vmssind DOUBLE PRECISION for vdsind DOUBLE PRECISION for vmdsind	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vssind REAL for vmssind DOUBLE PRECISION for vdsind DOUBLE PRECISION for vmdsind	Pointer to an array containing the output vector *y*.

Description

The v?Sind function computes the sine of vector elements multiplied by $\pi/180$. For an argument x, the function computes $\sin(\pi * x/180)$.

Special values for Real Function v?Sind(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
±∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Application Notes

If arguments abs(a_i) ≤ 2^{24} for single precision or abs(a_i) ≤ 2^{52} for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

Sin Computes sine of vector elements.

Sinpi Computes the sine of vector elements multiplied by *π*.

v?Tand

Computes the tangent of vector elements multiplied by π/180.

Syntax

```
call vstand (n, a, y)
call vstandi(n, a, inca, y, incy)
call vmstand (n, a, y, mode)
call vmstandi(n, a, inca, y, incy, mode)
call vdtand (n, a, y)
call vdtandi(n, a, inca, y, incy)
call vmdtand (n, a, y, mode)
call vmdtandi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vstand REAL for vmstand DOUBLE PRECISION for vdtand DOUBLE PRECISION for vmdtand	Pointer to the array containing the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description

y	`REAL` for `vstand`
	`REAL` for `vmstand`
	`DOUBLE PRECISION` for `vdtand`
	`DOUBLE PRECISION` for `vmdtand`
		Pointer to an array containing the output vector *y*.

Description

The `v?Tand` function computes the tangent of vector elements multiplied by $\pi/180$. For an argument x, the function computes $\tan(\pi*x/180)$.

Special values for Real Function v?Tand(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
±∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

The copysign(x, y) function returns the first vector argument x with the sign changed to match that of the second argument y.

Application Notes

If arguments $\text{abs}(a_i) \le 2^{38}$ for single precision or $\text{abs}(a_i) \le 2^{67}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

Tan Computes tangent of vector elements.
Tanpi Computes the tangent of vector elements multiplied by π.

Hyperbolic Functions

v?Cosh

Computes hyperbolic cosine of vector elements.

Syntax

```
call vscosh( n, a, y )
call vscoshi(n, a, inca, y, incy)
call vmscosh( n, a, y, mode )
call vmscoshi(n, a, inca, y, incy, mode)
call vdcosh( n, a, y )
call vdcoshi(n, a, inca, y, incy)
call vmdcosh( n, a, y, mode )
```

```
call vmdcoshi(n, a, inca, y, incy, mode)
call vccosh( n, a, y )
call vccoshi(n, a, inca, y, incy)
call vmccosh( n, a, y, mode )
call vmccoshi(n, a, inca, y, incy, mode)
call vzcosh( n, a, y )
call vzcoshi(n, a, inca, y, incy)
call vmzcosh( n, a, y, mode )
call vmzcoshi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdcosh, vmdcosh COMPLEX for vccosh, vmccosh DOUBLE COMPLEX for vzcosh, vmzcosh REAL, INTENT(IN) for vscosh, vmscosh DOUBLE PRECISION, INTENT(IN) for vdcosh, vmdcosh COMPLEX, INTENT(IN) for vccosh, vmccosh DOUBLE COMPLEX, INTENT(IN) for vzcosh, vmzcosh	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Cosh Function

Data Type	Threshold Limitations on Input Parameters
single precision	-Ln(FLT_MAX)-Ln2 <*a*[i] < Ln(FLT_MAX)+Ln2
double precision	-Ln(DBL_MAX)-Ln2 <*a*[i] < Ln(DBL_MAX)+Ln2

Precision overflow thresholds for the complex v?Cosh function are beyond the scope of this document.

Output Parameters

Name	Type	Description

y	`DOUBLE PRECISION` for `vdcosh`, `vmdcosh`
	`COMPLEX` for `vccosh`, `vmccosh`
	`DOUBLE COMPLEX` for `vzcosh`, `vmzcosh`
	`REAL, INTENT(OUT)` for `vscosh`, `vmscosh`
	`DOUBLE PRECISION, INTENT(OUT)` for `vdcosh`, `vmdcosh`
	`COMPLEX, INTENT(OUT)` for `vccosh`, `vmccosh`
	`DOUBLE COMPLEX, INTENT(OUT)` for `vzcosh`, `vmzcosh`
		Array that specifies the output vector *y*.

Description

The `v?Cosh` function computes hyperbolic cosine of vector elements.

Special Values for Real Function v?Cosh(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
X < -overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
+∞	+∞		
-∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Cosh(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+∞ +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN-i·0 INVALID	QNAN+i·0 INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN INVALID	QNAN +i·QNAN
+i·Y	+∞·Cos(Y)- i·∞·Sin(Y)					+∞·CIS(Y)	QNAN +i·QNAN
+i·0	+∞-i·0		+1-i·0	+1+i·0		+∞+i·0	QNAN+i·0
-i·0	+∞+i·0		+1+i·0	+1-i·0		+∞-i·0	QNAN-i·0

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
-i·Y	+∞·Cos(Y)- i·∞·Sin(Y)					+∞·CIS(Y)	QNAN +i·QNAN
-i·∞	+∞ +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN+i·0 INVALID	QNAN-i·0 INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN INVALID	QNAN +i·QNAN
+i·NAN	+∞ +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN- i·QNAN	QNAN +i·QNAN	+∞ +i·QNAN	QNAN +i·QNAN

Notes:

- raises the `INVALID` exception when the real or imaginary part of the argument is `SNAN`
- raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when `RE(z), IM(z)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- `Cosh(CONJ(z))=CONJ(Cosh(z))`
- `Cosh(-z)=Cosh(z).`

v?Sinh

Computes hyperbolic sine of vector elements.

Syntax

```
call vssinh( n, a, y )
call vssinhi(n, a, inca, y, incy)
call vmssinh( n, a, y, mode )
call vmssinhi(n, a, inca, y, incy, mode)
call vdsinh( n, a, y )
call vdsinhi(n, a, inca, y, incy)
call vmdsinh( n, a, y, mode )
call vmdsinhi(n, a, inca, y, incy, mode)
call vcsinh( n, a, y )
call vcsinhi(n, a, inca, y, incy)
call vmcsinh( n, a, y, mode )
call vmcsinhi(n, a, inca, y, incy, mode)
call vzsinh( n, a, y )
call vzsinhi(n, a, inca, y, incy)
call vmzsinh( n, a, y, mode )
call vmzsinhi(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdsinh, vmdsinh COMPLEX for vcsinh, vmcsinh DOUBLE COMPLEX for vzsinh, vmzsinh REAL, INTENT(IN) for vssinh, vmssinh DOUBLE PRECISION, INTENT(IN) for vdsinh, vmdsinh COMPLEX, INTENT(IN) for vcsinh, vmcsinh DOUBLE COMPLEX, INTENT(IN) for vzsinh, vmzsinh	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Sinh Function

Data Type	Threshold Limitations on Input Parameters
single precision	-Ln(FLT_MAX)-Ln2 <*a*[i] < Ln(FLT_MAX)+Ln2
double precision	-Ln(DBL_MAX)-Ln2 <*a*[i] < Ln(DBL_MAX)+Ln2

Precision overflow thresholds for the complex v?Sinh function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdsinh, vmdsinh COMPLEX for vcsinh, vmcsinh DOUBLE COMPLEX for vzsinh, vmzsinh REAL, INTENT(OUT) for vssinh, vmssinh DOUBLE PRECISION, INTENT(OUT) for vdsinh, vmdsinh COMPLEX, INTENT(OUT) for vcsinh, vmcsinh	Array that specifies the output vector *y*.

Name	Type	Description
	DOUBLE COMPLEX, INTENT(OUT) for vzsinh, vmzsinh	

Description

The v?Sinh function computes hyperbolic sine of vector elements.

Special Values for Real Function v?Sinh(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
X < -overflow	-∞	VML_STATUS_OVERFLOW	OVERFLOW
+∞	+∞		
-∞	-∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Sinh(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-∞+i·QNAN INVALID	QNAN +i·QNAN INVALID	-0+i·QNAN INVALID	+0+i·QNAN INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN INVALID	QNAN +i·QNAN
+i·Y	-∞·Cos(Y)+ i·∞·Sin(Y)					+∞·CIS(Y)	QNAN +i·QNAN
+i·0	-∞+i·0		-0+i·0	+0+i·0		+∞+i·0	QNAN+i·0
-i·0	-∞-i·0		-0-i·0	+0-i·0		+∞-i·0	QNAN-i·0
-i·Y	-∞·Cos(Y)+ i·∞·Sin(Y)					+∞·CIS(Y)	QNAN +i·QNAN
-i·∞	-∞+i·QNAN INVALID	QNAN +i·QNAN INVALID	-0+i·QNAN INVALID	+0+i·QNAN INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN INVALID	QNAN +i·QNAN
+i·NAN	-∞+i·QNAN	QNAN +i·QNAN	-0+i·QNAN	+0+i·QNAN	QNAN +i·QNAN	+∞ +i·QNAN	QNAN +i·QNAN

Notes:

- raises the INVALID exception when the real or imaginary part of the argument is SNAN
- raises the OVERFLOW exception and sets the VM Error Status to VML_STATUS_OVERFLOW in the case of overflow, that is, when RE(z), IM(z) are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- Sinh(CONJ(z))=CONJ(Sinh(z))
- Sinh(-z)=-Sinh(z).

v?Tanh

Computes hyperbolic tangent of vector elements.

Syntax

```
call vstanh( n, a, y )
call vstanhi(n, a, inca, y, incy)
call vmstanh( n, a, y, mode )
call vmstanhi(n, a, inca, y, incy, mode)
call vdtanh( n, a, y )
call vdtanhi(n, a, inca, y, incy)
call vmdtanh( n, a, y, mode )
call vmdtanhi(n, a, inca, y, incy, mode)
call vctanh( n, a, y )
call vctanhi(n, a, inca, y, incy)
call vmctanh( n, a, y, mode )
call vmctanhi(n, a, inca, y, incy, mode)
call vztanh( n, a, y )
call vztanhi(n, a, inca, y, incy)
call vmztanh( n, a, y, mode )
call vmztanhi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdtanh, vmdtanh COMPLEX for vctanh, vmctanh DOUBLE COMPLEX for vztanh, vmztanh REAL, INTENT(IN) for vstanh, vmstanh DOUBLE PRECISION, INTENT(IN) for vdtanh, vmdtanh COMPLEX, INTENT(IN) for vctanh, vmctanh	Array that specifies the input vector *a*.

Name	Type	Description
	DOUBLE COMPLEX, INTENT(IN) for vztanh, vmztanh	
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdtanh, vmdtanh COMPLEX for vctanh, vmctanh DOUBLE COMPLEX for vztanh, vmztanh REAL, INTENT(OUT) for vstanh, vmstanh DOUBLE PRECISION, INTENT(OUT) for vdtanh, vmdtanh COMPLEX, INTENT(OUT) for vctanh, vmctanh DOUBLE COMPLEX, INTENT(OUT) for vztanh, vmztanh	Array that specifies the output vector *y*.

Description

The v?Tanh function computes hyperbolic tangent of vector elements.

Special Values for Real Function v?Tanh(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+1	
-∞	-1	
QNAN	QNAN	
SNAN	QNAN	INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Tanh(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-1+i·0	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+1+i·0	QNAN +i·QNAN

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·Y	-1+i·0·Tan(Y)					+1+i·0·Tan(Y)	QNAN +i·QNAN
+i·0	-1+i·0		-0+i·0	+0+i·0		+1+i·0	QNAN+i·0
-i·0	-1-i·0		-0-i·0	+0-i·0		+1-i·0	QNAN-i·0
-i·Y	-1+i·0·Tan(Y)					+1+i·0·Tan(Y)	QNAN +i·QNAN
-i·∞	-1-i·0	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+1-i·0	QNAN +i·QNAN
+i·NAN	-1+i·0	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	+1+i·0	QNAN +i·QNAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- `Tanh(CONJ(z))=CONJ(Tanh(z))`
- `Tanh(-z)=-Tanh(z)`.

v?Acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

Syntax

```
call vsacosh( n, a, y )
call vsacoshi(n, a, inca, y, incy)
call vmsacosh( n, a, y, mode )
call vmsacoshi(n, a, inca, y, incy, mode)
call vdacosh( n, a, y )
call vdacoshi(n, a, inca, y, incy)
call vmdacosh( n, a, y, mode )
call vmdacoshi(n, a, inca, y, incy, mode)
call vcacosh( n, a, y )
call vcacoshi(n, a, inca, y, incy)
call vmcacosh( n, a, y, mode )
call vmcacoshi(n, a, inca, y, incy, mode)
call vzacosh( n, a, y )
call vzacoshi(n, a, inca, y, incy)
call vmzacosh( n, a, y, mode )
call vmzacoshi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdacosh, vmdacosh COMPLEX for vcacosh, vmcacosh DOUBLE COMPLEX for vzacosh, vmzacosh REAL, INTENT(IN) for vsacosh, vmsacosh DOUBLE PRECISION, INTENT(IN) for vdacosh, vmdacosh COMPLEX, INTENT(IN) for vcacosh, vmcacosh DOUBLE COMPLEX, INTENT(IN) for vzacosh, vmzacosh	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdacosh, vmdacosh COMPLEX for vcacosh, vmcacosh DOUBLE COMPLEX for vzacosh, vmzacosh REAL, INTENT(OUT) for vsacosh, vmsacosh DOUBLE PRECISION, INTENT(OUT) for vdacosh, vmdacosh COMPLEX, INTENT(OUT) for vcacosh, vmcacosh DOUBLE COMPLEX, INTENT(OUT) for vzacosh, vmzacosh	Array that specifies the output vector *y*.

Description

The `v?Acosh` function computes inverse hyperbolic cosine (nonnegative) of vector elements.

Special Values for Real Function v?Acosh(x)

Argument	Result	VM Error Status	Exception
+1	+0		
X < +1	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Acosh(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i \cdot \frac{3\pi}{4}$	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	+∞+i·π					+∞+i·0	QNAN +i·QNAN
+i·0	+∞+i·π		+0+i·π/2	+0+i·π/2		+∞+i·0	QNAN +i·QNAN
-i·0	+∞+i·π		+0+i·π/2	+0+i·π/2		+∞+i·0	QNAN +i·QNAN
-i·Y	+∞+i·π					+∞+i·0	QNAN +i·QNAN
-i·∞	$+\infty - i \cdot \frac{3\pi}{4}$	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	+∞ +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	+∞ +i·QNAN	QNAN +i·QNAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- `Acosh(CONJ(z))=CONJ(Acosh(z))`.

v?Asinh

Computes inverse hyperbolic sine of vector elements.

Syntax

```
call vsasinh( n, a, y )
call vsasinhi(n, a, inca, y, incy)
call vmsasinh( n, a, y, mode )
```

```
call vmsasinhi(n, a, inca, y, incy, mode)
call vdasinh( n, a, y )
call vdasinhi(n, a, inca, y, incy)
call vmdasinh( n, a, y, mode )
call vmdasinhi(n, a, inca, y, incy, mode)
call vcasinh( n, a, y )
call vcasinhi(n, a, inca, y, incy)
call vmcasinh( n, a, y, mode )
call vmcasinhi(n, a, inca, y, incy, mode)
call vzasinh( n, a, y )
call vzasinhi(n, a, inca, y, incy)
call vmzasinh( n, a, y, mode )
call vmzasinhi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdasinh, vmdasinh COMPLEX for vcasinh, vmcasinh DOUBLE COMPLEX for vzasinh, vmzasinh REAL, INTENT(IN) for vsasinh, vmsasinh DOUBLE PRECISION, INTENT(IN) for vdasinh, vmdasinh COMPLEX, INTENT(IN) for vcasinh, vmcasinh DOUBLE COMPLEX, INTENT(IN) for vzasinh, vmzasinh	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdasinh, vmdasinh COMPLEX for vcasinh, vmcasinh DOUBLE COMPLEX for vzasinh, vmzasinh REAL, INTENT(OUT) for vsasinh, vmsasinh DOUBLE PRECISION, INTENT(OUT) for vdasinh, vmdasinh COMPLEX, INTENT(OUT) for vcasinh, vmcasinh DOUBLE COMPLEX, INTENT(OUT) for vzasinh, vmzasinh	Array that specifies the output vector *y*.

Description

The v?Asinh function computes inverse hyperbolic sine of vector elements.

Special Values for Real Function v?Asinh(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Asinh(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-∞+i·π/4	-∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	-∞+i·0					+∞+i·0	QNAN +i·QNAN
+i·0	+∞+i·0		+0+i·0	+0+i·0		+∞+i·0	QNAN +i·QNAN
-i·0	-∞-i·0		-0-i·0	+0-i·0		+∞-i·0	QNAN- i·QNAN
-i·Y	-∞-i·0					+∞-i·0	QNAN +i·QNAN
-i·∞	-∞-i·π/4	-∞-i·π/2	-∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·NAN	-∞+i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	+∞+i·QNAN	QNAN +i·QNAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- `Asinh(CONJ(z))=CONJ(Asinh(z))`
- `Asinh(-z)=-Asinh(z)`.

v?Atanh

Computes inverse hyperbolic tangent of vector elements.

Syntax

```
call vsatanh( n, a, y )
call vsatanhi(n, a, inca, y, incy)
call vmsatanh( n, a, y, mode )
call vmsatanhi(n, a, inca, y, incy, mode)
call vdatanh( n, a, y )
call vdatanhi(n, a, inca, y, incy)
call vmdatanh( n, a, y, mode )
call vmdatanhi(n, a, inca, y, incy, mode)
call vcatanh( n, a, y )
call vcatanhi(n, a, inca, y, incy)
call vmcatanh( n, a, y, mode )
call vmcatanhi(n, a, inca, y, incy, mode)
call vzatanh( n, a, y )
call vzatanhi(n, a, inca, y, incy)
call vmzatanh( n, a, y, mode )
call vmzatanhi(n, a, inca, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER, INTENT(IN)`	Specifies the number of elements to be calculated.

Name	Type	Description
a	DOUBLE PRECISION for vdatanh, vmdatanh COMPLEX for vcatanh, vmcatanh DOUBLE COMPLEX for vzatanh, vmzatanh REAL, INTENT(IN) for vsatanh, vmsatanh DOUBLE PRECISION, INTENT(IN) for vdatanh, vmdatanh COMPLEX, INTENT(IN) for vcatanh, vmcatanh DOUBLE COMPLEX, INTENT(IN) for vzatanh, vmzatanh	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdatanh, vmdatanh COMPLEX for vcatanh, vmcatanh DOUBLE COMPLEX for vzatanh, vmzatanh REAL, INTENT(OUT) for vsatanh, vmsatanh DOUBLE PRECISION, INTENT(OUT) for vdatanh, vmdatanh COMPLEX, INTENT(OUT) for vcatanh, vmcatanh DOUBLE COMPLEX, INTENT(OUT) for vzatanh, vmzatanh	Array that specifies the output vector *y*.

Description

The v?Atanh function computes inverse hyperbolic tangent of vector elements.

Special Values for Real Function v?Atanh(x)

Argument	Result	VM Error Status	Exception
+1	+∞	VML_STATUS_SING	ZERODIVIDE
-1	-∞	VML_STATUS_SING	ZERODIVIDE

Argument	Result	VM Error Status	Exception
|X| > 1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Special Value Notations for the conventions used in the table below.

Special Values for Complex Function v?Atanh(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-0+i·π/2	-0+i·π/2	-0+i·π/2	+0+i·π/2	+0+i·π/2	+0+i·π/2	+0+i·π/2
+i·Y	-0+i·π/2					+0+i·π/2	QNAN +i·QNAN
+i·0	-0+i·π/2		-0+i·0	+0+i·0		+0+i·π/2	QNAN +i·QNAN
-i·0	-0-i·π/2		-0-i·0	+0-i·0		+0-i·π/2	QNAN- i·QNAN
-i·Y	-0-i·π/2					+0-i·π/2	QNAN +i·QNAN
-i·∞	-0-i·π/2	-0-i·π/2	-0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2
+i·NAN	-0+i·QNAN	QNAN +i·QNAN	-0+i·QNAN	+0+i·QNAN	QNAN +i·QNAN	+0+i·QNAN	QNAN +i·QNAN

Notes:

- `Atanh(+-1+-i*0)=+-∞+-i*0`, and `ZERODIVIDE` exception is raised
- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- `Atanh(CONJ(z))=CONJ(Atanh(z))`
- `Atanh(-z)=-Atanh(z)`.

Special Functions

v?Erf

Computes the error function value of vector elements.

Syntax

```
call vserf( n, a, y )
call vserfi(n, a, inca, y, incy)
call vmserf( n, a, y, mode )
call vmserfi(n, a, inca, y, incy, mode)
call vderf( n, a, y )
call vderfi(n, a, inca, y, incy)
call vmderf( n, a, y, mode )
```

```
call vmderfi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vderf, vmderf REAL, INTENT(IN) for vserf, vmserf DOUBLE PRECISION, INTENT(IN) for vderf, vmderf	Array, specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vderf, vmderf REAL, INTENT(OUT) for vserf, vmserf DOUBLE PRECISION, INTENT(OUT) for vderf, vmderf	Array, specifies the output vector *y*.

Description

The Erf function computes the error function values for elements of the input vector *a* and writes them to the output vector *y*.

The error function is defined as given by:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

$$1.\quad \mathrm{erfc}(x) = 1 - \mathrm{erf}(x),$$

where `erfc` is the complementary error function.

$$2.\quad \Phi(x) = \frac{1}{2}\,\mathrm{erf}\,(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_0^x \exp\,(-t^2/2)dt$$

is the cumulative normal distribution function.

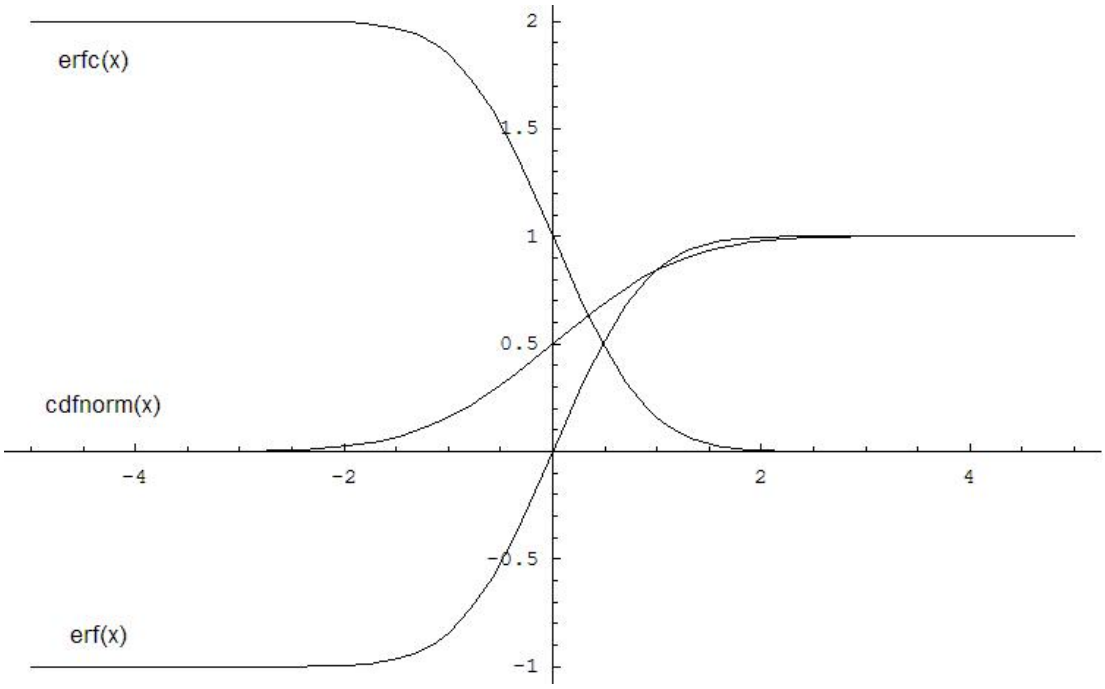
$$3.\quad \Phi^{-1}(x) = \sqrt{2}\,\mathrm{erf}^{-1}(2x - 1),$$

where $\Phi^{-1}(x)$ and $\mathrm{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\mathrm{erf}(x)$ respectively.

The following figure illustrates the relationships among `Erf` family functions (`Erf`, `Erfc`, `CdfNorm`).

__border__top

Erf Family Functions Relationship



Useful relations for these functions:

$$\mathrm{erf}(x) + \mathrm{erfc}(x) = 1$$

$$\mathrm{cdfnorm}(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) = 1 - \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

Special Values for Real Function v?Erf(x)

Argument	Result	Exception
+∞	+1	
-∞	-1	
QNAN	QNAN	
SNAN	QNAN	INVALID

See Also

Erfc

CdfNorm

v?Erfc

Computes the complementary error function value of vector elements.

Syntax

```
call vserfc( n, a, y )
call vserfci(n, a, inca, y, incy)
call vmserfc( n, a, y, mode )
call vmserfci(n, a, inca, y, incy, mode)
call vderfc( n, a, y )
call vderfci(n, a, inca, y, incy)
call vmderfc( n, a, y, mode )
call vmderfci(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vderfc, vmderfc REAL, INTENT(IN) for vserfc, vmserfc DOUBLE PRECISION, INTENT(IN) for vderfc, vmderfc	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description

y	`DOUBLE PRECISION` for `vderfc`, `vmderfc`
	`REAL, INTENT(OUT)` for `vserfc`, `vmserfc`
	`DOUBLE PRECISION, INTENT(OUT)` for `vderfc, vmderfc`
		Array that specifies the output vector *y*.

Description

The `Erfc` function computes the complementary error function values for elements of the input vector *a* and writes them to the output vector *y*.

The complementary error function is defined as follows:

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} e^{-t^2} dt .$$

Useful relations:

1. $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$.

2. $\Phi(x) = \frac{1}{2} \mathrm{erf}\left(x/\sqrt{2}\right)$,

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{0}^{x} \exp\left(-t^2/2\right) dt$$

is the cumulative normal distribution function.

3. $\Phi^{-1}(x) = \sqrt{2}\, \mathrm{erf}^{-1}(2x - 1)$,

where $\Phi^{-1}(x)$ and $\mathrm{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\mathrm{erf}(x)$ respectively.

See also Figure "Erf Family Functions Relationship" in `Erf` function description for `Erfc` function relationship with the other functions of `Erf` family.

Special Values for Real Function v?Erfc(x)

Argument	Result	VM Error Status	Exception
X > underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW

Argument	Result	VM Error Status	Exception
+∞	+0		
-∞	+2		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

Erf

CdfNorm

v?CdfNorm

Computes the cumulative normal distribution function values of vector elements.

Syntax

```
call vscdfnorm( n, a, y )
call vscdfnormi(n, a, inca, y, incy)
call vmscdfnorm( n, a, y, mode )
call vmscdfnormi(n, a, inca, y, incy, mode)
call vdcdfnorm( n, a, y )
call vdcdfnormi(n, a, inca, y, incy)
call vmdcdfnorm( n, a, y, mode )
call vmdcdfnormi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdcdfnorm, vmdcdfnorm REAL, INTENT(IN) for vscdfnorm, vmscdfnorm DOUBLE PRECISION, INTENT(IN) for vdcdfnorm, vmdcdfnorm	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description

y	`DOUBLE PRECISION` for `vdcdfnorm`, `vmdcdfnorm`
	`REAL, INTENT(OUT)` for `vscdfnorm`, `vmscdfnorm`
	`DOUBLE PRECISION, INTENT(OUT)` for `vdcdfnorm, vmdcdfnorm`
		Array that specifies the output vector *y*.

Description

The `CdfNorm` function computes the cumulative normal distribution function values for elements of the input vector *a* and writes them to the output vector *y*.

The cumulative normal distribution function is defined as given by:

$$\mathrm{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt .$$

Useful relations:

$$\mathrm{cdfnorm}(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) = 1 - \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

where `Erf` and `Erfc` are the error and complementary error functions.

See also Figure "Erf Family Functions Relationship" in `Erf` function description for `CdfNorm` function relationship with the other functions of `Erf` family.

Special Values for Real Function v?CdfNorm(x)

Argument	Result	VM Error Status	Exception
X < underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
+∞	+1		
-∞	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

`Erf`

`Erfc`

v?ErfInv

Computes inverse error function value of vector elements.

Syntax

```
call vserfinv( n, a, y )
call vserfinvi(n, a, inca, y, incy)
call vmserfinv( n, a, y, mode )
call vmserfinvi(n, a, inca, y, incy, mode)
call vderfinv( n, a, y )
call vderfinvi(n, a, inca, y, incy)
call vmderfinv( n, a, y, mode )
call vmderfinvi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vderfinv, vmderfinv REAL, INTENT(IN) for vserfinv, vmserfinv DOUBLE PRECISION, INTENT(IN) for vderfinv, vmderfinv	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vderfinv, vmderfinv REAL, INTENT(OUT) for vserfinv, vmserfinv DOUBLE PRECISION, INTENT(OUT) for vderfinv, vmderfinv	Array that specifies the output vector *y*.

Description

The ErfInv function computes the inverse error function values for elements of the input vector *a* and writes them to the output vector *y*

$y = \mathrm{erf}^{-1}(a)$,

where $\mathrm{erf}(x)$ is the error function defined as given by:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

$$1.\quad \mathrm{erf}^{-1}(x) = \mathrm{erfc}^{-1}(1 - x),$$

where erfc is the complementary error function.

$$2.\quad \Phi(x) = \frac{1}{2}\,\mathrm{erf}\,(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp\,(-t^2/2)dt$$

is the cumulative normal distribution function.

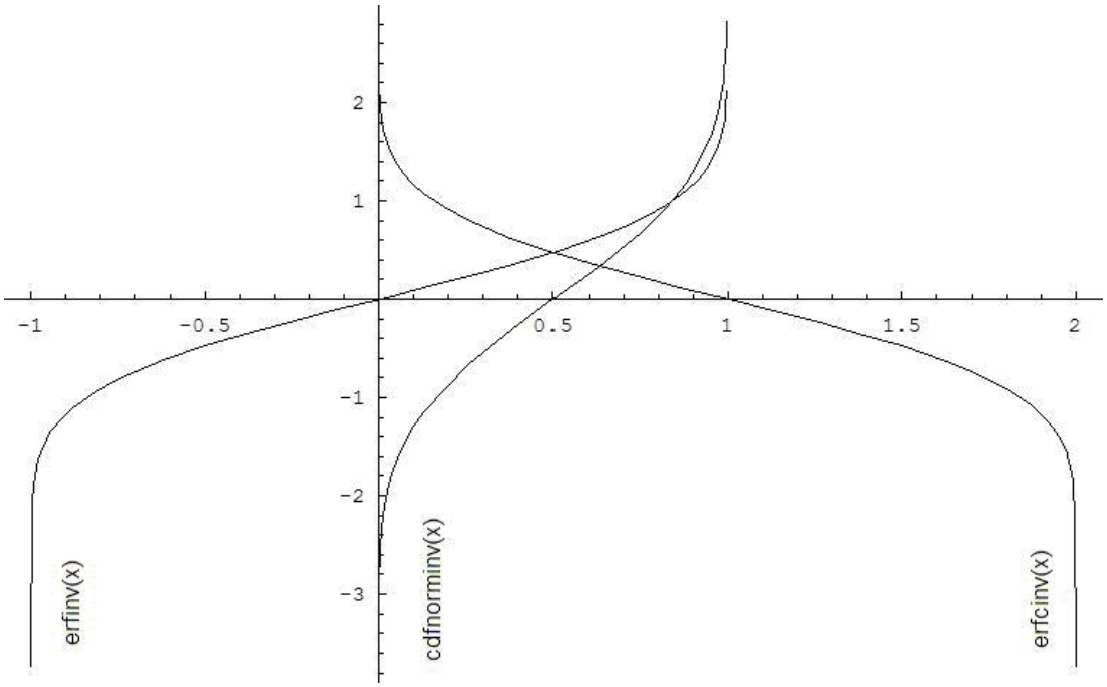
$$3.\quad \Phi^{-1}(x) = \sqrt{2}\,\mathrm{erf}^{-1}(2x - 1),$$

where $\Phi^{-1}(x)$ and $\mathrm{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\mathrm{erf}(x)$ respectively.

Figure "ErfInv Family Functions Relationship" illustrates the relationships among `ErfInv` family functions (`ErfInv`, `ErfcInv`, `CdfNormInv`).

__border__top

ErfInv Family Functions Relationship



Useful relations for these functions:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

Special Values for Real Function v?ErfInv(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+1	+∞	VML_STATUS_SING	ZERODIVIDE
-1	-∞	VML_STATUS_SING	ZERODIVIDE
|X| > 1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

ErfcInv

CdfNormInv

v?ErfcInv

Computes the inverse complementary error function value of vector elements.

Syntax

```
call vserfcinv( n, a, y )
call vserfcinvi(n, a, inca, y, incy)
call vmserfcinv( n, a, y, mode )
call vmserfcinvi(n, a, inca, y, incy, mode)
call vderfcinv( n, a, y )
call vderfcinvi(n, a, inca, y, incy)
call vmderfcinv( n, a, y, mode )
call vmderfcinvi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vderfcinv, vmderfcinv REAL, INTENT(IN) for vserfcinv, vmserfcinv DOUBLE PRECISION, INTENT(IN) for vderfcinv, vmderfcinv	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vderfcinv, vmderfcinv REAL, INTENT(OUT) for vserfcinv, vmserfcinv DOUBLE PRECISION, INTENT(OUT) for vderfcinv, vmderfcinv	Array that specifies the output vector *y*.

Description

The ErfcInv function computes the inverse complimentary error function values for elements of the input vector *a* and writes them to the output vector *y*.

The inverse complementary error function is defined as given by:

$$\mathrm{erfcinv}(x) = \mathrm{erfinv}(1 - x)$$

$$\mathrm{erfinv}(x) = \mathrm{erf}^{-1}(x)$$

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

where `erf(x)` denotes the error function and `erfinv(x)` denotes the inverse error function.

See also Figure "ErfInv Family Functions Relationship" in `ErfInv` function description for `ErfcInv` function relationship with the other functions of `ErfInv` family.

Special Values for Real Function v?ErfcInv(x)

Argument	Result	VM Error Status	Exception
+1	+0		
+2	-∞	VML_STATUS_SING	ZERODIVIDE
-0	+∞	VML_STATUS_SING	ZERODIVIDE
+0	+∞	VML_STATUS_SING	ZERODIVIDE
X < -0	QNAN	VML_STATUS_ERRDOM	INVALID
X > +2	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

`ErfInv`

`CdfNormInv`

v?CdfNormInv

Computes the inverse cumulative normal distribution function values of vector elements.

Syntax

```
call vscdfnorminv( n, a, y )
```

```
call vscdfnorminvi(n, a, inca, y, incy)
```

```
call vmscdfnorminv( n, a, y, mode )
call vmscdfnorminvi(n, a, inca, y, incy, mode)
call vdcdfnorminv( n, a, y )
call vdcdfnorminvi(n, a, inca, y, incy)
call vmdcdfnorminv( n, a, y, mode )
call vmdcdfnorminvi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdcdfnorminv, vmdcdfnorminv REAL, INTENT(IN) for vscdfnorminv, vmscdfnorminv DOUBLE PRECISION, INTENT(IN) for vdcdfnorminv, vmdcdfnorminv	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdcdfnorminv, vmdcdfnorminv REAL, INTENT(OUT) for vscdfnorminv, vmscdfnorminv DOUBLE PRECISION, INTENT(OUT) for vdcdfnorminv, vmdcdfnorminv	Array that specifies the output vector *y*.

Description

The CdfNormInv function computes the inverse cumulative normal distribution function values for elements of the input vector *a* and writes them to the output vector *y*.

The inverse cumulative normal distribution function is defined as given by:

$$\mathrm{CdfNormInv}(x) = \mathrm{CdfNorm}^{-1}(x),$$

where `CdfNorm(x)` denotes the cumulative normal distribution function.

Useful relations:

$$\mathrm{cdfnorminv}(x) = \sqrt{2}\,\mathrm{erfinv}(2x - 1) = \sqrt{2}\,\mathrm{erfcinv}(2 - 2x)$$

where `erfinv(x)` denotes the inverse error function and `erfcinv(x)` denotes the inverse complementary error functions.

See also Figure "ErfInv Family Functions Relationship" in `ErfInv` function description for `CdfNormInv` function relationship with the other functions of `ErfInv` family.

Special Values for Real Function v?CdfNormInv(x)

Argument	Result	VM Error Status	Exception
+0.5	+0		
+1	+∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
+0	-∞	VML_STATUS_SING	ZERODIVIDE
X < -0	QNAN	VML_STATUS_ERRDOM	INVALID
X > +1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

`ErfInv`

`ErfcInv`

v?LGamma

Computes the natural logarithm of the absolute value of gamma function for vector elements.

Syntax

```
call vslgamma( n, a, y )
call vslgammai(n, a, inca, y, incy)
call vmslgamma( n, a, y, mode )
call vmslgammai(n, a, inca, y, incy, mode)
call vdlgamma( n, a, y )
call vdlgammai(n, a, inca, y, incy)
call vmdlgamma( n, a, y, mode )
call vmdlgammai(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdlgamma, vmdlgamma REAL, INTENT(IN) for vslgamma, vmslgamma DOUBLE PRECISION, INTENT(IN) for vdlgamma, vmdlgamma	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdlgamma, vmdlgamma REAL, INTENT(OUT) for vslgamma, vmslgamma DOUBLE PRECISION, INTENT(OUT) for vdlgamma, vmdlgamma	Array that specifies the output vector *y*.

Description

The v?LGamma function computes the natural logarithm of the absolute value of gamma function for elements of the input vector *a* and writes them to the output vector *y*. Precision overflow thresholds for the v?LGamma function are beyond the scope of this document. If the result does not meet the target precision, the function raises the OVERFLOW exception and sets the VM Error Status to VML_STATUS_OVERFLOW.

Special Values for Real Function v?LGamma(x)

Argument	Result	VM Error Status	Exception
+1	+0		
+2	+0		
+0	+∞	VML_STATUS_SING	ZERODIVIDE
-0	+∞	VML_STATUS_SING	ZERODIVIDE
negative integer	+∞	VML_STATUS_SING	ZERODIVIDE
-∞	+∞		
+∞	+∞		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
QNAN	QNAN		

Argument	Result	VM Error Status	Exception
SNAN	QNAN		INVALID

v?TGamma

Computes the gamma function of vector elements.

Syntax

```
call vstgamma( n, a, y )
call vstgammai(n, a, inca, y, incy)
call vmstgamma( n, a, y, mode )
call vmstgammai(n, a, inca, y, incy, mode)
call vdtgamma( n, a, y )
call vdtgammai(n, a, inca, y, incy)
call vmdtgamma( n, a, y, mode )
call vmdtgammai(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdtgamma, vmdtgamma REAL, INTENT(IN) for vstgamma, vmstgamma DOUBLE PRECISION, INTENT(IN) for vdtgamma, vmdtgamma	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdtgamma, vmdtgamma REAL, INTENT(OUT) for vstgamma, vmstgamma	Array that specifies the output vector *y*.

Name	Type	Description
	DOUBLE PRECISION, INTENT(OUT) for vdtgamma, vmdtgamma	

Description

The v?TGamma function computes the gamma function for elements of the input vector *a* and writes them to the output vector *y*. Precision overflow thresholds for the v?TGamma function are beyond the scope of this document. If the result does not meet the target precision, the function raises the OVERFLOW exception and sets the VM Error Status to VML_STATUS_OVERFLOW.

Special Values for Real Function v?TGamma(x)

Argument	Result	VM Error Status	Exception
+0	+∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
negative integer	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
QNAN	QNAN		
SNAN	QNAN		INVALID

v?ExpInt1

Computes the exponential integral of vector elements.

Syntax

```
call vsexpint1( n, a, y )
call vsexpint1i(n, a, inca, y, incy)
call vmsexpint1( n, a, y, mode )
call vmsexpint1i(n, a, inca, y, incy, mode)
call vdexpint1( n, a, y )
call vdexpint1i(n, a, inca, y, incy)
call vdexpint1( n, a, y, mode )
call vmdexpint1i(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	REAL (KIND=4), INTENT(IN) for vsexpint1, vmsexpint1	Array that specifies the input vector *a*.

Name	Type	Description
	`REAL (KIND=8), INTENT(IN)` for `vdexpint1, vmdexpint1`	
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	`INTEGER(KIND=8), INTENT(IN)`	Overrides global VM mode setting for this function call. See `vmlSetMode` for possible values and their description.

Output Parameters

Name	Type	Description
y	`REAL (KIND=4), INTENT(OUT)` for `vsexpint1, vmsexpint1`	Array that specifies the output vector *y*.
	`REAL (KIND=8), INTENT(OUT)` for `vdexpint1, vmdexpint1`	

Description

The `v?ExpInt1` function computes the exponential integral E_1 of vector elements.

For positive real values *x*, this can be written as:

$$E_1(x) = \int_x^\infty \frac{e^{-t}}{t}\,\mathrm{d}t = \int_1^\infty \frac{e^{-xt}}{t}\,\mathrm{d}t.$$

For negative real values *x*, the result is defined as `NAN`.

Special Values for Real Function v?ExpInt1(x)

Argument	Result	VM Error Status	Exception
x < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+∞	VML_STATUS_SING	ZERODIVIDE
-0	+∞	VML_STATUS_SING	ZERODIVIDE
+∞	+0		
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Rounding Functions

v?Floor

Computes an integer value rounded towards minus infinity for each vector element.

Syntax

```
call vsfloor( n, a, y )
call vsfloori(n, a, inca, y, incy)
call vmsfloor( n, a, y, mode )
call vmsfloori(n, a, inca, y, incy, mode)
call vdfloor( n, a, y )
```

```
call vdfloori(n, a, inca, y, incy)
call vmdfloor( n, a, y, mode )
call vmdfloori(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

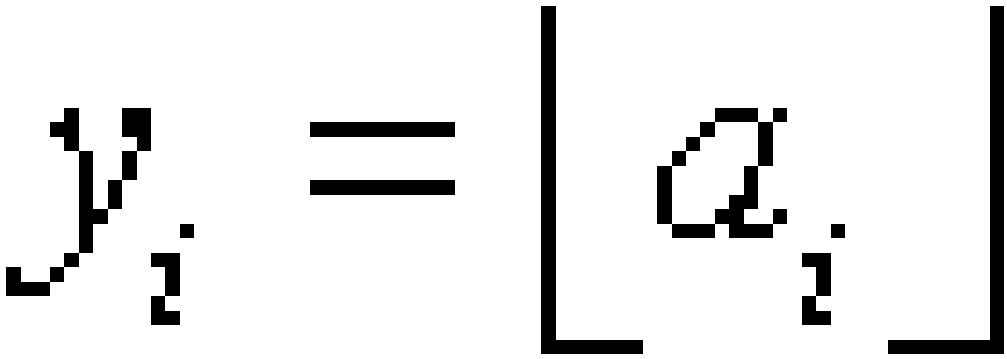
Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdfloor, vmdfloor REAL, INTENT(IN) for vsfloor, vmsfloor DOUBLE PRECISION, INTENT(IN) for vdfloor, vmdfloor	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdfloor, vmdfloor REAL, INTENT(OUT) for vsfloor, vmsfloor DOUBLE PRECISION, INTENT(OUT) for vdfloor, vmdfloor	Array that specifies the output vector *y*.

Description

The function computes an integer value rounded towards minus infinity for each vector element.



Special Values for Real Function v?Floor(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Ceil

Computes an integer value rounded towards plus infinity for each vector element.

Syntax

```
call vsceil( n, a, y )
call vsceili(n, a, inca, y, incy)
call vmsceil( n, a, y, mode )
call vmsceili(n, a, inca, y, incy, mode)
call vdceil( n, a, y )
call vdceili(n, a, inca, y, incy)
call vmdceil( n, a, y, mode )
call vmdceili(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.

Name	Type	Description

a	`DOUBLE PRECISION` for `vdceil`, `vmdceil`
	`REAL, INTENT(IN)` for `vsceil`, `vmsceil`
	`DOUBLE PRECISION, INTENT(IN)` for `vdceil, vmdceil`
		Array that specifies the input vector *a*.

inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	`INTEGER(KIND=8), INTENT(IN)`	Overrides global VM mode setting for this function call. See `vmlSetMode` for possible values and their description.

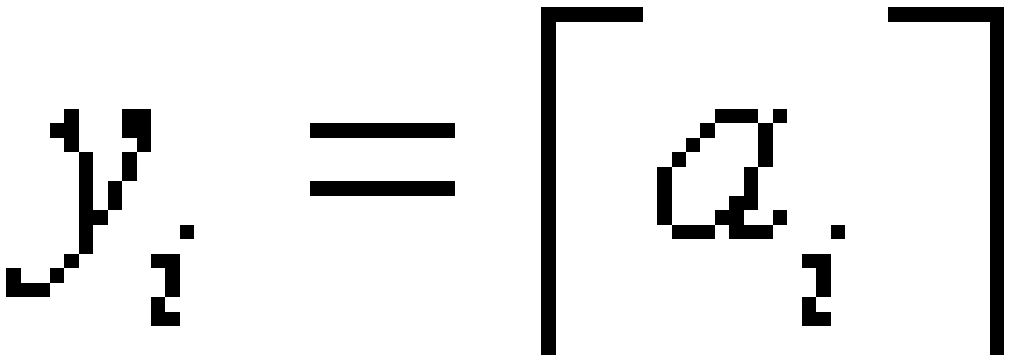
Output Parameters

Name	Type	Description

y	`DOUBLE PRECISION` for `vdceil`, `vmdceil`
	`REAL, INTENT(OUT)` for `vsceil`, `vmsceil`
	`DOUBLE PRECISION, INTENT(OUT)` for `vdceil, vmdceil`
		Array that specifies the output vector *y*.

Description

The function computes an integer value rounded towards plus infinity for each vector element.



Special Values for Real Function v?Ceil(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID

Argument	Result	Exception
QNAN	QNAN	

v?Trunc

Computes an integer value rounded towards zero for each vector element.

Syntax

```
call vstrunc( n, a, y )
call vstrunci(n, a, inca, y, incy)
call vmstrunc( n, a, y, mode )
call vmstrunci(n, a, inca, y, incy, mode)
call vdtrunc( n, a, y )
call vdtrunci(n, a, inca, y, incy)
call vmdtrunc( n, a, y, mode )
call vmdtrunci(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdTrunc, vmdTrunc REAL, INTENT(IN) for vsTrunc, vmsTrunc DOUBLE PRECISION, INTENT(IN) for vdTrunc, vmdTrunc	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdTrunc, vmdTrunc REAL, INTENT(OUT) for vsTrunc, vmsTrunc	Array that specifies the output vector *y*.

Name	Type	Description
	`DOUBLE PRECISION, INTENT(OUT)` for `vdTrunc, vmdTrunc`	

Description

The function computes an integer value rounded towards zero for each vector element.

$$a_i \geq 0, y_i = \lfloor a_i \rfloor$$

$$a_i < 0, y_i = \lceil a_i \rceil$$

Special Values for Real Function v?Trunc(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Round

Computes a value rounded to the nearest integer for each vector element.

Syntax

```
call vsround( n, a, y )
call vsroundi(n, a, inca, y, incy)
call vmsround( n, a, y, mode )
call vmsroundi(n, a, inca, y, incy, mode)
call vdround( n, a, y )
call vdroundi(n, a, inca, y, incy)
call vmdround( n, a, y, mode )
call vmdroundi(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdround, vmdround REAL, INTENT(IN) for vsround, vmsround DOUBLE PRECISION, INTENT(IN) for vdround, vmdround	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdround, vmdround REAL, INTENT(OUT) for vsround, vmsround DOUBLE PRECISION, INTENT(OUT) for vdround, vmdround	Array that specifies the output vector *y*.

Description

The function computes a value rounded to the nearest integer for each vector element. Input elements that are halfway between two consecutive integers are always rounded away from zero regardless of the rounding mode.

Special Values for Real Function v?Round(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?NearbyInt

Computes a rounded integer value in the current rounding mode for each vector element.

Syntax

```
call vsnearbyint( n, a, y )
call vsnearbyinti(n, a, inca, y, incy)
call vmsnearbyint( n, a, y, mode )
call vmsnearbyinti(n, a, inca, y, incy, mode)
call vdnearbyint( n, a, y )
call vdnearbyinti(n, a, inca, y, incy)
call vmdnearbyint( n, a, y, mode )
call vmdnearbyinti(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdnearbyint, vmdnearbyint REAL, INTENT(IN) for vsnearbyint, vmsnearbyint DOUBLE PRECISION, INTENT(IN) for vdnearbyint, vmdnearbyint	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdnearbyint, vmdnearbyint REAL, INTENT(OUT) for vsnearbyint, vmsnearbyint DOUBLE PRECISION, INTENT(OUT) for vdnearbyint, vmdnearbyint	Array that specifies the output vector *y*.

Description

The v?NearbyInt function computes a rounded integer value in a current rounding mode for each vector element.

Special Values for Real Function v?NearbyInt(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Rint

Computes a rounded integer value in the current rounding mode.

Syntax

```
call vsrint( n, a, y )
call vsrinti(n, a, inca, y, incy)
call vmsrint( n, a, y, mode )
call vmsrinti(n, a, inca, y, incy, mode)
call vdrint( n, a, y )
call vdrinti(n, a, inca, y, incy)
call vmdrint( n, a, y, mode )
call vmdrinti(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdrint, vmdrint REAL, INTENT(IN) for vsrint, vmsrint DOUBLE PRECISION, INTENT(IN) for vdrint, vmdrint	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdrint, vmdrint REAL, INTENT(OUT) for vsrint, vmsrint DOUBLE PRECISION, INTENT(OUT) for vdrint, vmdrint	Array that specifies the output vector *y*.

Description

The v?Rint function computes a rounded floating-point integer value using the current rounding mode for each vector element.

The rounding mode affects the results computed for inputs that fall between consecutive integers. For example:

- f(0.5) = 0, for rounding modes set to round to nearest round toward zero or to minus infinity.
- f(0.5) = 1, for rounding modes set to plus infinity.
- f(-1.5) = -2, for rounding modes set to round to nearest or to minus infinity.
- f(-1.5) = -1, for rounding modes set to round toward zero or to plus infinity.

Special Values for Real Function v?Rint(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Modf

Computes a truncated integer value and the remaining fraction part for each vector element.

Syntax

```
call vsmodf( n, a, y, z )
call vsmodfi(n, a, inca, y, incy, z, incz)
call vmsmodf( n, a, y, z, mode )
call vmsmodfi(n, a, inca, y, incy, z, incz, mode)
call vdmodf( n, a, y, z )
call vdmodfi(n, a, inca, y, incy, z, incz)
call vmdmodf( n, a, y, z, mode )
call vmdmodfi(n, a, inca, y, incy, z, incz, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdmodf, vmdmodf REAL, INTENT(IN) for vsmodf, vmsmodf DOUBLE PRECISION, INTENT(IN) for vdmodf, vmdmodf	Array, specifies the input vector *a*.
inca, *incy*, *incz*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *y*, and *z*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y, *z*	DOUBLE PRECISION for vdmodf, vmdmodf REAL, INTENT(OUT) for vsmodf, vmsmodf DOUBLE PRECISION, INTENT(OUT) for vdmodf, vmdmodf	Array, specifies the output vector *y* and *z*.

Description

The function computes a truncated integer value and the remaining fraction part for each vector element.

$$a_i \geq 0, \begin{cases} y_i = \lfloor a_i \rfloor \\ z_i = a_i - \lfloor a_i \rfloor \end{cases}$$

$$a_i < 0, \begin{cases} y_i = \lceil a_i \rceil \\ z_i = a_i - \lceil a_i \rceil \end{cases}$$

Special Values for Real Function v?Modf(x)

Argument	Result: *y*(*i*)	Result: *z*(*i*)	Exception
+0	+0	+0	
-0	-0	-0	
+∞	+∞	+0	
-∞	-∞	-0	
SNAN	QNAN	QNAN	INVALID
QNAN	QNAN	QNAN	

v?Frac

Computes a signed fractional part for each vector element.

Syntax

```
call vsfrac( n, a, y )
call vsfraci(n, a, inca, y, incy)
call vmsfrac( n, a, y, mode )
call vmsfraci(n, a, inca, y, incy, mode)
call vdfrac( n, a, y )
call vdfraci(n, a, inca, y, incy)
call vmdfrac( n, a, y, mode )
call vmdfraci(n, a, inca, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdfrac, vmdfrac REAL, INTENT(IN) for vsfrac, vmsfrac DOUBLE PRECISION, INTENT(IN) for vdfrac, vmdfrac	Array that specifies the input vector *a*.
inca, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a* and *y*.
mode	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdfrac, vmdfrac REAL, INTENT(OUT) for vsfrac, vmsfrac DOUBLE PRECISION, INTENT(OUT) for vdfrac, vmdfrac	Array that specifies the output vector *y*.

Description

The function computes a signed fractional part for each vector element.

$$y_i = \begin{cases} a_i - \lfloor a_i \rfloor, a_i \geq 0 \\ a_i - \lceil a_i \rceil, a_i < 0 \end{cases}$$

Special Values for Real Function v?Frac(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+0	
-∞	-0	
SNAN	QNAN	INVALID
QNAN	QNAN	

VM Pack/Unpack Functions

This section describes VM functions that convert vectors with unit increment to and from vectors with positive increment indexing, vector indexing, and mask indexing (see Appendix "Vector Arguments in VM" for details on vector indexing methods).

The table below lists available VM Pack/Unpack functions, together with data types and indexing methods associated with them.

VM Pack/Unpack Functions

Function Short Name	Data Types	Indexing Methods	Description
v?pack	s, d, c, z	I,V,M	Gathers elements of arrays, indexed by different methods.
v?unpack	s, d, c, z	I,V,M	Scatters vector elements to arrays with different indexing.

See Also

Appendix "Vector Arguments in VM"

v?Pack

Copies elements of an array with specified indexing to a vector with unit increment.

Syntax

```
call vspacki( n, a, inca, y )
call vspackv( n, a, ia, y )
call vspackm( n, a, ma, y )
call vdpacki( n, a, inca, y )
call vdpackv( n, a, ia, y )
call vdpackm( n, a, ma, y )
call vcpacki( n, a, inca, y )
call vcpackv( n, a, ia, y )
call vcpackm( n, a, ma, y )
call vzpacki( n, a, inca, y )
call vzpackv( n, a, ia, y )
call vzpackm( n, a, ma, y )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdpacki, vdpackv, vdpackm COMPLEX for vcpacki, vcpackv, vcpackm DOUBLE COMPLEX for vzpacki, vzpackv, vzpackm REAL, INTENT(IN) for vspacki, vspackv, vspackm DOUBLE PRECISION, INTENT(IN) for vdpacki, vdpackv, vdpackm COMPLEX, INTENT(IN) for vcpacki, vcpackv, vcpackm DOUBLE COMPLEX, INTENT(IN) for vzpacki, vzpackv, vzpackm	Array, DIMENSION at least (1 + (*n*-1)***inca*) for v?packi, Array, DIMENSION at least max(*n*,max(*ia*[j])), j=0, …, *n*-1 for v?packv, Array, DIMENSION at least *n* for v?packm. Specifies the input vector *a*.
inca	INTEGER, INTENT(IN) for vspacki, vdpacki, vcpacki, vzpacki	Specifies the increment for the elements of *a*.
ia	FORTRAN 77: INTEGER for vspackv, vdpackv, vcpackv, vzpackv INTEGER, INTENT(IN) for vspackv, vdpackv, vcpackv, vzpackv	Array, DIMENSION at least *n*. Specifies the index vector for the elements of *a*.
ma	FORTRAN 77: INTEGER for vspackm, vdpackm, vcpackm, vzpackm Fortran 90: INTEGER, INTENT(IN) for vspackm, vdpackm, vcpackm, vzpackm	Array, DIMENSION at least *n*, Specifies the mask vector for the elements of *a*.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdpacki, vdpackv, vdpackm COMPLEX for vcpacki, vcpackv, vcpackm	Array, DIMENSION at least *n*. Specifies the output vector *y*.

Name	Type	Description
	DOUBLE COMPLEX for vzpacki, vzpackv, vzpackm	
	REAL, INTENT(OUT) for vspacki, vspackv, vspackm	
	DOUBLE PRECISION, INTENT(OUT) for vdpacki, vdpackv, vdpackm	
	COMPLEX, INTENT(OUT) for vcpacki, vcpackv, vcpackm	
	DOUBLE COMPLEX, INTENT(OUT) for vzpacki, vzpackv, vzpackm	

v?Unpack

Copies elements of a vector with unit increment to an array with specified indexing.

Syntax

```
call vsunpacki( n, a, y, incy )
call vsunpackv( n, a, y, iy )
call vsunpackm( n, a, y, my )
call vdunpacki( n, a, y, incy )
call vdunpackv( n, a, y, iy )
call vdunpackm( n, a, y, my )
call vcunpacki( n, a, y, incy )
call vcunpackv( n, a, y, iy )
call vcunpackm( n, a, y, my )
call vzunpacki( n, a, y, incy )
call vzunpackv( n, a, y, iy )
call vzunpackm( n, a, y, my )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdunpacki, vdunpackv, vdunpackm COMPLEX for vcunpacki, vcunpackv, vcunpackm	Array, DIMENSION at least *n*. Specifies the input vector *a*.

Name	Type	Description
	DOUBLE COMPLEX for vzunpacki, vzunpackv, vznpackm REAL, INTENT(IN) for vsunpacki, vsunpackv, vsunpackm DOUBLE PRECISION, INTENT(IN) for vdunpacki, vdunpackv, vdunpackm COMPLEX, INTENT(IN) for vcunpacki, vcunpackv, vcunpackm DOUBLE COMPLEX, INTENT(IN) for vzunpacki, vzunpackv, vzunpackm	
incy	INTEGER, INTENT(IN) for vsunpacki, vdunpacki, vcunpacki, vzunpacki	Specifies the increment for the elements of *y*.
iy	INTEGER, INTENT(IN) for vsunpackv, vdunpackv, vcunpackv, vzunpackv	Array, DIMENSION at least *n*. Specifies the index vector for the elements of *y*.
my	INTEGER, INTENT(IN) for vsunpackm, vdunpackm, vcunpackm, vzunpackm	Array, DIMENSION at least *n*, Specifies the mask vector for the elements of *y*.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdunpacki, vdunpackv, vdunpackm COMPLEX, INTENT(IN) for vcunpacki, vcunpackv, vcunpackm DOUBLE COMPLEX, INTENT(IN) for vzunpacki, vzunpackv, vzunpackm REAL, INTENT(OUT) for vsunpacki, vsunpackv, vsunpackm DOUBLE PRECISION, INTENT(OUT) for vdunpacki, vdunpackv, vdunpackm COMPLEX, INTENT(OUT) for vcunpacki, vcunpackv, vcunpackm DOUBLE COMPLEX, INTENT(OUT) for vzunpacki, vzunpackv, vzunpackm	Array, DIMENSION for v?unpacki, at least (1 + (*n*-1)***incy*) for v?unpackv, at least max(*n*,max(*iy*[j])),j=0,..., *n*-1 for v?unpackm, at least *n* for v?UnpackI, at least (1 + (*n*-1)***incy*) for v?UnpackV, at least max(*n*,max(*ia*[j])),j=0,..., *n*-1, for v?UnpackM, at least *n*.

VM Service Functions

The VM Service functions enable you to set/get the accuracy mode and error code. These functions are available both in the Fortran and C interfaces. The table below lists available VM Service functions and their short description.

VM Service Functions

Function Short Name	Description
vmlsetmode	Sets the VM mode
vmlgetmode	Gets the VM mode
vmlseterrstatus	Sets the VM Error Status
vmlgeterrstatus	Gets the VM Error Status
vmlclearerrstatus	Clears the VM Error Status
vmlseterrorcallback	Sets the additional error handler callback function
vmlgeterrorcallback	Gets the additional error handler callback function
vmlclearerrorcallback	Deletes the additional error handler callback function

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

vmlSetMode

Sets a new mode for VM functions according to the mode parameter and stores the previous VM mode to oldmode.

Syntax

```
oldmode = vmlsetmode( mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
mode	INTEGER(KIND=8), INTENT(IN)	Specifies the VM mode to be set.

Output Parameters

Name	Type	Description
oldmode	INTEGER*8 INTEGER(KIND=8)	Specifies the former VM mode.

Description

The vmlSetMode function sets a new mode for VM functions according to the *mode* parameter and stores the previous VM mode to *oldmode*. The mode change has a global effect on all the VM functions within a thread.

NOTE
You can override the global mode setting and change the mode for a given VM function call by using a respective vm[s,d]<Func> variant of the function.

The *mode* parameter is designed to control accuracy, handling of denormalized numbers, and error handling. Table "Values of the *mode* Parameter" lists values of the *mode* parameter. You can obtain all other possible values of the *mode* parameter from the *mode* parameter values by a using bitwise `OR ( | )` operation to combine one value for accuracy, one value for handling of denormalized numbers, and one value for error control options. The default value of the *mode* parameter is `VML_HA | VML_FTZDAZ_CURRENT | VML_ERRMODE_DEFAULT`.

The `VML_FTZDAZ_ON` mode is specifically designed to improve the performance of computations that involve denormalized numbers at the cost of reasonable accuracy loss. This mode changes the numeric behavior of the functions: denormalized input values are treated as zeros (`DAZ` = denormals-are-zero) and denormalized results are flushed to zero (`FTZ` = flush-to-zero). Accuracy loss may occur if input and/or output values are close to denormal range.

Values of the *mode* Parameter

Value of *mode*	Description
Accuracy Control	
`VML_HA`	high accuracy versions of VM functions
`VML_LA`	low accuracy versions of VM functions
`VML_EP`	enhanced performance accuracy versions of VM functions
Denormalized Numbers Handling Control	
`VML_FTZDAZ_ON`	Faster processing of denormalized inputs is enabled.
`VML_FTZDAZ_OFF`	Faster processing of denormalized inputs is disabled.
`VML_FTZDAZ_CURRENT`	Keep the current CPU settings for denormalized inputs.
Error Mode Control	
`VML_ERRMODE_IGNORE`	On computation error, VM Error status is updated, but otherwise no action is set. Cannot be combined with other `VML_ERRMODE` settings.
`VML_ERRMODE_NOERR`	On computation error, VM Error status is not updated and no action is set. Cannot be combined with other `VML_ERRMODE` settings.
`VML_ERRMODE_STDERR`	On error, the error text information is written to *stderr*.
`VML_ERRMODE_EXCEPT`	On error, an exception is raised.
`VML_ERRMODE_CALLBACK`	On error, an additional error handler function is called.
`VML_ERRMODE_DEFAULT`	On error, an exception is raised and an additional error handler function is called.

Examples

The following example shows how to set low accuracy, fast processing for denormalized numbers and `stderr` error mode:

```
oldmode = vmlsetmode( VML_LA )
call vmlsetmode( IOR(VML_LA, VML_FTZDAZ_ON, VML_ERRMODE_STDERR) )
```

vmlgetmode

Gets the VM mode.

Syntax

```
mod = vmlgetmode()
```

Include Files

- mkl_vml.f90

Output Parameters

Name	Type	Description
mod	INTEGER	Specifies the packed *mode* parameter.

Description

The function `vmlgetmode` returns the VM *mode* parameter that controls accuracy, handling of denormalized numbers, and error handling options. The *mod* variable value is a combination of the values listed in the table "Values of the *mode* Parameter". You can obtain these values using the respective mask from the table "Values of Mask for the *mode* Parameter".

Values of Mask for the *mode* Parameter

Value of mask	Description
VML_ACCURACY_MASK	Specifies mask for accuracy *mode* selection.
VML_FTZDAZ_MASK	Specifies mask for FTZDAZ *mode* selection.
VML_ERRMODE_MASK	Specifies mask for error *mode* selection.

See example below:

Examples

```
mod = vmlgetmode()
accm = IAND(mod, VML_ACCURACY_MASK)
denm = IAND(mod, VML_FTZDAZ_MASK)
errm = IAND(mod, VML_ERRMODE_MASK)
```

vmlSetErrStatus

Sets the new VM Error Status according to err and stores the previous VM Error Status to olderrSets the global VM Status according to new values and returns the previous VM Status.

Syntax

```
olderr = vmlseterrstatus( status )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
status	INTEGER, INTENT(IN)	Specifies the VM error status to be set.

Output Parameters

Name	Type	Description
olderr	INTEGER	Specifies the former VM error status.

Description

Table "Values of the VM Status" lists possible values of the *err* parameter.

Values of the VM Status

Status	Description
Successful Execution	
VML_STATUS_OK	The execution was completed successfully.
Warnings	
VML_STATUS_ACCURACYWARNING	The execution was completed successfully in a different accuracy mode.
Errors	
VML_STATUS_BADSIZE	The function does not support the preset accuracy mode. The Low Accuracy mode is used instead.
VML_STATUS_BADMEM	NULL pointer is passed.
VML_STATUS_ERRDOM	At least one of array values is out of a range of definition.
VML_STATUS_SING	At least one of the input array values causes a divide-by-zero exception or produces an invalid (QNaN) result.
VML_STATUS_OVERFLOW	An overflow has happened during the calculation process.
VML_STATUS_UNDERFLOW	An underflow has happened during the calculation process.

Examples

```
olderr = vmlSetErrStatus( VML_STATUS_OK );
olderr = vmlSetErrStatus( VML_STATUS_ERRDOM );
olderr = vmlSetErrStatus( VML_STATUS_UNDERFLOW );
```

vmlgeterrstatus

Gets the VM Error Status.

Syntax

```
err = vmlgeterrstatus( )
```

Include Files

- mkl_vml.f90

Output Parameters

Name	Type	Description
err	INTEGER	Specifies the VM error status.

vmlclearerrstatus

Sets the VM Error Status to VML_STATUS_OK *and stores the previous VM Error Status to olderr.*

Syntax

olderr = vmlclearerrstatus( )

Include Files

- mkl_vml.f90

Output Parameters

Name	Type	Description
olderr	INTEGER	Specifies the former VM error status.

vmlSetErrorCallBack

Sets the additional error handler callback function and gets the old callback function.

Syntax

oldcallback = vmlseterrorcallback(*callback*)

Include Files

- mkl_vml.f90

Input Parameters

Name		Description

callback	Address of the callback function.	The callback function has the following format:
		`INTEGER FUNCTION ERRFUNC(par)`
		`TYPE (ERROR_STRUCTURE) par`
		`! ...`
		`! user error processing`
		`! ...`
		`ERRFUNC = 0`
		`! if ERRFUNC= 0 - standard VM error handler`
		`! is called after the callback`
		`! if ERRFUNC != 0 - standard VM error handler`
		`! is not called`
		`END`
		The passed error structure is defined as follows:
		`TYPE ERROR_STRUCTURE SEQUENCE`
		`INTEGER*4 ICODE`
		`INTEGER*4 IINDEX`
		`REAL*8 DBA1`
		`REAL*8 DBA2`
		`REAL*8 DBR1`
		`REAL*8 DBR2`
		`CHARACTER(64) CFUNCNAME`
		`INTEGER*4 IFUNCNAMELEN`
		`REAL*8 DBA1IM`
		`REAL*8 DBA2IM`
		`REAL*8 DBR1IM`
		`REAL*8 DBR2IM`
		`END TYPE ERROR_STRUCTURE`

Output Parameters

Name	Type	Description
oldcallback	INTEGER	Address of the former callback function.

NOTE
This function does not have a FORTRAN 77 interface due to the use of internal structures.

Description

The callback function is called on each VM mathematical function error if `VML_ERRMODE_CALLBACK` error mode is set (see "Values of the *mode* Parameter").

Use the `vmlSetErrorCallBack()` function if you need to define your own callback function instead of default empty callback function.

The input structure for a callback function contains the following information about the error encountered:

- the input value that caused an error
- location (array index) of this value
- the computed result value
- error code
- name of the function in which the error occurred.

You can insert your own error processing into the callback function. This may include correcting the passed result values in order to pass them back and resume computation. The standard error handler is called after the callback function only if it returns *0*.

vmlGetErrorCallBack

Gets the additional error handler callback function.

Syntax

callback = vmlgeterrorcallback( )

Include Files

- mkl_vml.f90

Output Parameters

Name	Description
callback	Address of the callback function

vmlClearErrorCallBack

Deletes the additional error handler callback function and retrieves the former callback function.

Syntax

oldcallback = vmlclearerrorcallback( )

Include Files

- mkl_vml.f90

Output Parameters

Name	Type	Description
oldcallback	INTEGER	Address of the former callback function

Miscellaneous VM Functions

v?CopySign

Returns vector of elements of one argument with signs changed to match other argument elements.

Syntax

```
call vscopysign (n, a, y)
call vscopysigni(n, a, inca, b, incb, y, incy)
call vmscopysign (n, a, y, mode)
call vmscopysigni(n, a, inca, b, incb, y, incy, mode)
call vdcopysign (n, a, y)
call vdcopysigni(n, a, inca, b, incb, y, incy)
call vmdcopysign (n, a, y, mode)
call vmdcopysigni(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a	REAL for vscopysign REAL for vmscopysign DOUBLE PRECISION for vdcopysign DOUBLE PRECISION for vmdcopysign	Pointer to the array containing the input vector *a*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vscopysign REAL for vmscopysign DOUBLE PRECISION for vdcopysign DOUBLE PRECISION for vmdcopysign	Pointer to an array containing the output vector *y*.

Description

The v?CopySign function returns the first vector argument elements with the sign changed to match the sign of the second vector argument's corresponding elements.

v?NextAfter

Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector.

Syntax

```
call vsnextafter (n, a, b, y)
call vsnextafteri(n, a, inca, b, incb, y, incy)
call vmsnextafter (n, a, b, y, mode)
call vmsnextafteri(n, a, inca, b, incb, y, incy, mode)
call vdnextafter (n, a, b, y)
call vdnextafteri(n, a, inca, b, incb, y, incy)
call vmdnextafter (n, a, b, y, mode)
call vmdnextafteri(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a, *b*	REAL for vsnextafter REAL for vmsnextafter DOUBLE PRECISION for vdnextafter DOUBLE PRECISION for vmdnextafter	Pointers to the arrays containing the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vsnextafter	Pointer to an array containing the output vector *y*.

Name	Type	Description
	`REAL` for `vmsnextafter`	
	`DOUBLE PRECISION` for `vdnextafter`	
	`DOUBLE PRECISION` for `vmdnextafterjjssd`	

Description

The `v?NextAfter` function returns a vector containing the next representable floating-point values following the first vector argument elements in the direction of the second vector argument's corresponding elements.

Special cases:

Overflow	The function raises overflow and inexact floating-point exceptions and sets `VML_STATUS_OVERFLOW` if an input vector argument element is finite and the corresponding result vector element value is infinite.
Underflow	The function raises underflow and inexact floating-point exceptions and sets `VML_STATUS_UNDERFLOW` if a result vector element value is subnormal or zero, and different from the corresponding input vector argument element.

Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

v?Fdim

Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise.

Syntax

```
call vsfdim (n, a, b, y)
call vsfdimi(n, a, inca, b, incb, y, incy)
call vmsfdim (n, a, b, y, mode)
call vmsfdimi(n, a, inca, b, incb, y, incy, mode)
call vdfdim (n, a, b, y)
call vdfdimi(n, a, inca, b, incb, y, incy)
call vmdfdim (n, a, b, y, mode)
call vmdfdimi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER`	Specifies the number of elements to be calculated.

Name	Type	Description

a, *b*	`REAL` for `vsfdim`
	`REAL` for `vmsfdim`
	`DOUBLE PRECISION` for `vdfdim`
	`DOUBLE PRECISION` for `vmdfdim`
		Pointers to the arrays containing the input vectors *a* and *b*.

inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	`INTEGER (KIND=8)`	Overrides the global VM *mode* setting for this function call. See `vmlSetMode` for possible values and their description.

Output Parameters

Name	Type	Description

y	`REAL` for `vsfdim`
	`REAL` for `vmsfdim`
	`DOUBLE PRECISION` for `vdfdim`
	`DOUBLE PRECISION` for `vmdfdimjjssd`
		Pointer to an array containing the output vector *y*.

Description

The `v?Fdim` function returns a vector containing the differences of the corresponding elements of the first and second vector arguments if the first element is larger, and +0 otherwise.

Special values for Real Function v?Fdim(x, y)

Argument 1	Argument 2	Result	VM Error Status	Exception
any	QNAN	QNAN		
any	SNAN	QNAN		INVALID
QNAN	any	QNAN		
SNAN	any	QNAN		INVALID

v?Fmax

Returns the larger of each pair of elements of the two vector arguments.

Syntax

```
call vsfmax (n, a, b, y)
call vsfmaxi(n, a, inca, b, incb, y, incy)
call vmsfmax (n, a, b, y, mode)
call vmsfmaxi(n, a, inca, b, incb, y, incy, mode)
call vdfmax (n, a, b, y)
call vdfmaxi(n, a, inca, b, incb, y, incy)
call vmdfmax (n, a, b, y, mode)
call vmdfmaxi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a, *b*	REAL for vsfmax REAL for vmsfmax DOUBLE PRECISION for vdfmax DOUBLE PRECISION for vmdfmax	Pointers to the arrays containing the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vsfmax REAL for vmsfmax DOUBLE PRECISION for vdfmax DOUBLE PRECISION for vmdfmaxjjssd	Pointer to an array containing the output vector *y*.

Description

The v?Fmax function returns a vector with element values equal to the larger value from each pair of corresponding elements of the two vectors *a* and *b*: if $a_i < b_i$ v?Fmax returns b_i, otherwise v?Fmax returns a_i.

Special values for Real Function v?Fmax(x, y)

Argument 1	Argument 2	Result	VM Error Status	Exception
a_i not NAN	NAN	a_i		
NAN	b_i not NAN	b_i		
NAN	NAN	NAN		

See Also

Fmin Returns the smaller of each pair of elements of the two vector arguments.

MaxMag Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

v?Fmin

Returns the smaller of each pair of elements of the two vector arguments.

Syntax

```
call vsfmin (n, a, b, y)
call vsfmini(n, a, inca, b, incb, y, incy)
call vmsfmin (n, a, b, y, mode)
call vmsfmini(n, a, inca, b, incb, y, incy, mode)
call vdfmin (n, a, b, y)
call vdfmini(n, a, inca, b, incb, y, incy)
call vmdfmin (n, a, b, y, mode)
call vmdfmini(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a, *b*	REAL for vsfmin REAL for vmsfmin DOUBLE PRECISION for vdfmin DOUBLE PRECISION for vmdfmin	Pointers to the arrays containing the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for vsfmin REAL for vmsfmin DOUBLE PRECISION for vdfmin DOUBLE PRECISION for vmdfminjjssd	Pointer to an array containing the output vector *y*.

Description

The v?Fmin function returns a vector with element values equal to the smaller value from each pair of corresponding elements of the two vectors *a* and *b*: if $b_i < a_i$ v?Fmin returns b_i, otherwise v?Fmin returns a_i.

Special values for Real Function v?Fmin(x, y)

Argument 1	Argument 2	Result	VM Error Status	Exception
a_i not NAN	NAN	a_i		

Argument 1	Argument 2	Result	VM Error Status	Exception
NAN	b_i not NAN	b_i		
NAN	NAN	NAN		

See Also

Fmax Returns the larger of each pair of elements of the two vector arguments.

MinMag Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

v?MaxMag

Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

Syntax

```
call vsmaxmag (n, a, b, y)
call vsmaxmagi(n, a, inca, b, incb, y, incy)
call vmsmaxmag (n, a, b, y, mode)
call vmsmaxmagi(n, a, inca, b, incb, y, incy, mode)
call vdmaxmag (n, a, b, y)
call vdmaxmagi(n, a, inca, b, incb, y, incy)
call vmdmaxmag (n, a, b, y, mode)
call vmdmaxmagi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER	Specifies the number of elements to be calculated.
a, *b*	REAL for vsmaxmag REAL for vmsmaxmag DOUBLE PRECISION for vdmaxmag DOUBLE PRECISION for vmdmaxmag	Pointers to the arrays containing the input vectors *a* and *b*.
inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	INTEGER (KIND=8)	Overrides the global VM *mode* setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description

y	`REAL` for `vsmaxmag`
	`REAL` for `vmsmaxmag`
	`DOUBLE PRECISION` for `vdmaxmag`
	`DOUBLE PRECISION` for `vmdmaxmagjjssd`
		Pointer to an array containing the output vector *y*.

Description

The `v?MaxMag` function returns a vector with element values equal to the element with the larger magnitude from each pair of corresponding elements of the two vectors *a* and *b*:

- If $|a_i| > |b_i|$ `v?MaxMag` returns a_i, otherwise `v?MaxMag` returns a_i.
- If $|b_i| > |a_i|$ `v?MaxMag` returns b_i, otherwise `v?MaxMag` returns a_i.
- Otherwise `v?MaxMag` behaves like `v?Fmax`.

Special values for Real Function v?MaxMag(x, y)

Argument 1	Argument 2	Result	VM Error Status	Exception
a_i not NAN	NAN	a_i		
NAN	b_i not NAN	b_i		
NAN	NAN	NAN		

See Also

MinMag Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

Fmax Returns the larger of each pair of elements of the two vector arguments.

v?MinMag

Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

Syntax

```
call vsminmag (n, a, b, y)
call vsminmagi(n, a, inca, b, incb, y, incy)
call vmsminmag (n, a, b, y, mode)
call vmsminmagi(n, a, inca, b, incb, y, incy, mode)
call vdminmag (n, a, b, y)
call vdminmagi(n, a, inca, b, incb, y, incy)
call vmdminmag (n, a, b, y, mode)
call vmdminmagi(n, a, inca, b, incb, y, incy, mode)
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	`INTEGER`	Specifies the number of elements to be calculated.

a, *b*	`REAL` for `vsminmag`
	`REAL` for `vmsminmag`
	`DOUBLE PRECISION` for `vdminmag`
	`DOUBLE PRECISION` for `vmdminmag`
		Pointers to the arrays containing the input vectors *a* and *b*.

inca, *incb*, *incy*	INTEGER, INTENT(IN)	Specifies increments for the elements of *a*, *b*, and *y*.
mode	`INTEGER (KIND=8)`	Overrides the global VM *mode* setting for this function call. See `vmlSetMode` for possible values and their description.

Output Parameters

Name	Type	Description

y	`REAL` for `vsminmag`
	`REAL` for `vmsminmag`
	`DOUBLE PRECISION` for `vdminmag`
	`DOUBLE PRECISION` for `vmdminmagjjssd`
		Pointer to an array containing the output vector *y*.

Description

The `v?MinMag` function returns a vector with element values equal to the element with the smaller magnitude from each pair of corresponding elements of the two vectors *a* and *b*:

- If $|a_i| < |b_i|$ `v?MaxMag` returns a_i, otherwise `v?MaxMag` returns a_i.
- If $|b_i| < |a_i|$ `v?MaxMag` returns b_i, otherwise `v?MaxMag` returns a_i.
- Otherwise `v?MaxMag` behaves like `v?Fmin`.

Special values for Real Function v?MinMag(x, y)

Argument 1	Argument 2	Result	VM Error Status	Exception
a_i not NAN	NAN	a_i		
NAN	b_i not NAN	b_i		
NAN	NAN	NAN		

See Also

MaxMag Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

Fmin Returns the smaller of each pair of elements of the two vector arguments.

Statistical Functions

Statistical functions in Intel® oneAPI Math Kernel Library (oneMKL) are known as the Vector Statistics (VS). They are designed for the purpose of

- generating vectors of pseudorandom, quasi-random, and non-deterministic random numbers
- performing mathematical operations of convolution and correlation
- computing basic statistical estimates for single and double precision multi-dimensional datasets

The corresponding functionality is described in the respective Random Number Generators, Convolution and Correlation, and Summary Statistics topics.

See VS performance data in the online VS Performance Data document available at https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html.

The basic notion in VS is a task. The task object is a data structure or descriptor that holds the parameters related to a specific statistical operation: random number generation, convolution and correlation, or summary statistics estimation. Such parameters can be an identifier of a random number generator, its internal state and parameters, data arrays, their shape and dimensions, an identifier of the operation and so forth. You can modify the VS task parameters using the VS service functions.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Random Number Generators

Intel® oneAPI Math Kernel Library (oneMKL) VS provides a set of routines implementing commonly used pseudorandom, quasi-random, or non-deterministic random number generators with continuous and discrete distribution. To improve performance, all these routines were developed using the calls to the highly optimized*Basic Random Number Generators* (BRNGs) and vector mathematical functions (VM, see "Vector Mathematical Functions").

VS provides interfaces both for Fortran and C languages. For users of the Fortran 90 or Fortran 95 language the `mkl_vsl.f90` header file is provided. The `mkl_vsl.fi`header file available in the previous versions of Intel® oneAPI Math Kernel Library (oneMKL) is retained for backward compatibility.All header files are found in the following directory:

```
${MKL}/include
```

The `mkl_vsl.f90` header is intended for use with the Fortran `include` clause and is compatible with both standard forms of F90/F95 sources - the free and 72-columns fixed forms. If you need to use the VS interface with 80- or 132-columns fixed form sources, you may add a new file to your project. That file is formatted as a 72-columns fixed-form source and consists of a single `include` clause as follows:

```
include 'mkl_vsl.f90'
```

This `include` clause causes the compiler to generate the module files `mkl_vsl.mod` and `mkl_vsl_type.mod`, which are used to process the Fortran use clauses referencing to the VS interface:

```
use mkl_vsl_type
```

```
use mkl_vsl
```

Because of this specific feature, you do not need to include the `mkl_vsl.f90` header into each source of your project. You only need to include the header into some of the sources. In any case, make sure that the sources that depend on the VS interface are compiled after those that include the header so that the module files `mkl_vsl.mod` and `mkl_vsl_type.mod` are generated prior to using them.

NOTE
For the Fortran interface, VS provides both a subroutine-style interface and a function-style interface. The default interface in this case is a function-style interface. The function-style interface, unlike the subroutine-style interface, allows the user to get error status of each routine. The subroutine-style interface is provided for backward compatibility only. To use the subroutine-style interface, manually include `mkl_vsl_subroutine.fi` file instead of `mkl_vsl.f90` by changing the line `include 'mkl_vsl.f90'` in `include\mkl.fi` with the line `include 'mkl_vsl_subroutine.fi'`.

All VS routines can be classified into three major categories:

- Transformation routines for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These routines indirectly call basic random number generators, which are pseudorandom, quasi-random, or non-deterministic random number generators. Detailed description of the generators can be found in Distribution Generators.
- Service routines to handle random number streams: create, initialize, delete, copy, save to a binary file, load from a binary file, get the index of a basic generator. The description of these routines can be found in Service Routines.
- Registration routines for basic pseudorandom generators and routines that obtain properties of the registered generators (see Advanced Service Routines).

The last two categories are referred to as service routines.

Random Number Generators Conventions

This document makes no specific differentiation between random, pseudorandom, and quasi-random numbers, nor between random, pseudorandom, and quasi-random number generators unless the context requires otherwise. For details, refer to the '*Random Numbers*' section in VS Notesdocument provided at the Intel® oneAPI Math Kernel Library (oneMKL) web page.

All generators of nonuniform distributions, both discrete and continuous, are built on the basis of the uniform distribution generators, called Basic Random Number Generators (BRNGs). The pseudorandom numbers with nonuniform distribution are obtained through an appropriate transformation of the uniformly distributed pseudorandom numbers. Such transformations are referred to as *generation methods*. For a given distribution, several generation methods can be used. See VS Notes for the description of methods available for each generator.

An RNG task determines environment in which random number generation is performed, in particular parameters of the BRNG and its internal state. Output of VS generators is a stream of random numbers that are used in Monte Carlo simulations. A *random stream descriptor* and a *random stream* are used as synonyms of an *RNG task* in the document unless the context requires otherwise.

The *random stream descriptor* specifies which BRNG should be used in a given transformation method. See the *Random Streams and RNGs in Parallel Computation* section of VS Notes.

The term *computational node* means a logical or physical unit that can process data in parallel.

Random Number Generators Mathematical Notation

The following notation is used throughout the text:

N	The set of natural numbers $N = \{1, 2, 3 \ldots\}$.
Z	The set of integers $Z = \{\ldots -3, -2, -1, 0, 1, 2, 3 \ldots\}$.
R	The set of real numbers.

$\lfloor a \rfloor$

The floor of *a* (the largest integer less than or equal to *a*).

⊕ or **xor**

Bitwise exclusive OR.

C_α^κ or $\binom{\alpha}{\kappa}$

Binomial coefficient or combination ($\alpha \in R$, $\alpha \geq 0$; $k \in N \cup \{0\}$).

$$C_\alpha^0 = 1$$

For $\alpha \geq k$ binomial coefficient is defined as

$$C_\alpha^\kappa = \frac{\alpha(\alpha - 1) \ldots (\alpha - \kappa + 1)}{\kappa!}$$

If $\alpha < k$, then

$$C_\alpha^\kappa = 0$$

$\Phi(x)$

Cumulative Gaussian distribution function

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) dy$$

defined over $-\infty < x < +\infty$.

$\Phi(-\infty) = 0$, $\Phi(+\infty) = 1$.

$\Gamma(\alpha)$

The complete gamma function

$$\Gamma(\alpha) = \int_0^\infty t^{\alpha-1} e^{-t} dt$$

where $\alpha > 0$.

B(*p*, *q*)

The complete beta function

$$B(p,q) = \int_0^1 t^{p-1}(1-t)^{q-1}dt$$

where $p>0$ and $q>0$.

LCG(*a*,*c*, *m*)	Linear Congruential Generator $x_{n+1} = (ax_n + c) \bmod m$, where *a* is called the *multiplier*, *c* is called the *increment*, and *m* is called the *modulus* of the generator.
MCG(*a*,*m*)	Multiplicative Congruential Generator $x_{n+1} = (ax_n) \bmod m$ is a special case of Linear Congruential Generator, where the increment *c* is taken to be 0.
GFSR(*p*, *q*)	Generalized Feedback Shift Register Generator $x_n = x_{n-p} \oplus x_{n-q}$.

Random Number Generators Naming Conventions

The names of the routines in VS random number generators are lowercase (`virnguniform`). The names are not case-sensitive.

The names of generator routines have the following structure:

`v<type of result>rng<distribution>`

where

- `v` is the prefix of a VS vector function.
- <*type of result*> is either `s`, `d`, or `i` and specifies one of the following types:

`s`	`REAL`
`d`	`DOUBLE PRECISION`
`i`	`INTEGER`

  Prefixes `s` and `d` apply to continuous distributions only, prefix `i` applies only to discrete case.

- `rng` indicates that the routine is a random generator.
- <*distribution*> specifies the type of statistical distribution.

Names of service routines follow the template below:

```
vsl<name>
```

where

- `vsl` is the prefix of a VS service function.
- <*name*> contains a short function name.

For a more detailed description of service routines, refer to Service Routines and Advanced Service Routines.

The prototype of each generator routine corresponding to a given probability distribution fits the following structure:

```
status = <function name>( method, stream, n, r, [<distribution parameters>] )
```

where

- *method* defines the method of generation. A detailed description of this parameter can be found in table "Values of <*method*> in *method* parameter". See below, where the structure of the *method* parameter name is explained.
- *stream* defines the descriptor of the random stream and must have a non-zero value. Random streams, descriptors, and their usage are discussed further in Random Streams and Service Routines.
- *n* defines the number of random values to be generated. If *n* is less than or equal to zero, no values are generated. Furthermore, if *n* is negative, an error condition is set.
- *r* defines the destination array for the generated numbers. The dimension of the array must be large enough to store at least *n* random numbers.
- *status* defines the error status of a VS routine. See Error Reporting for a detailed description of error status values.

Additional parameters included into <*distribution parameters*> field are individual for each generator routine and are described in detail in Distribution Generators.

To invoke a distribution generator, use a call to the respective VS routine. For example, to obtain a vector *r*, composed of *n* independent and identically distributed random numbers with normal (Gaussian) distribution, that have the mean value *a* and standard deviation *sigma*, write the following:

```
status = vsrnggaussian( method, stream, n, r, a, sigma )
```

The name of a *method* parameter has the following structure:

```
VSL_RNG_METHOD_method<distribution>_<method>
```

```
VSL_RNG_METHOD_<distribution>_<method>_ACCURATE
```

where

- <*distribution*> is the probability distribution.
- <*method*> is the method name.

Type of the name structure for the *method* parameter corresponds to fast and accurate modes of random number generation (see "Distribution Generators" and VS Notes for details).

Method names VSL_RNG_METHOD_<*distribution*>_<*method*>

and

VSL_RNG_METHOD_<*distribution*>_<*method*>_ACCURATE

should be used with

v<*precision*>Rng<*distribution*>

function only, where

- <*precision*> is

s	for single precision continuous distribution
d	for double precision continuous distribution
i	for discrete distribution

- <*distribution*> is the probability distribution.

is the probability distribution.Table "Values of <*method*> in *method* parameter" provides specific predefined values of the *method* name. The third column contains names of the functions that use the given method.

Values of <*method*> in *method* parameter

Method	Short Description	Functions
STD	Standard method. Currently there is only one method for these functions.	Uniform (continuous), Uniform (discrete),

Method	Short Description	Functions
		UniformBits, UniformBits32, UniformBits64
BOXMULLER	BOXMULLER generates normally distributed random number x thru the pair of uniformly distributed numbers u_1 and u_2 according to the formula: $x = \sqrt{-2 \ln u_1} \sin 2\pi u_2$	Gaussian, GaussianMV
BOXMULLER2	BOXMULLER2 generates normally distributed random numbers x_1 and x_2 thru the pair of uniformly distributed numbers u_1 and u_2 according to the formulas: $x_1 = \sqrt{-2 \ln u_1} \sin 2\pi u_2$ $x_2 = \sqrt{-2 \ln u_1} \cos 2\pi u_2$	Gaussian, GaussianMV, Lognormal
ICDF	Inverse cumulative distribution function method.	Exponential, Laplace, Weibull, Cauchy, Rayleigh, Gumbel, Bernoulli, Geometric, Gaussian, GaussianMV, Lognormal
GNORM	For $\alpha > 1$, a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \le \alpha < 1$, a gamma distributed random number is generated using rejection from Weibull distribution; for $\alpha < 0.6$, a gamma distributed random number is obtained using transformation of exponential power distribution; for $\alpha = 1$, gamma distribution is reduced to exponential distribution.	Gamma
CJA	For $min(p, q) > 1$, Cheng method is used; for $min(p, q) < 1$, Johnk method is used, if $q + K \cdot p^2 + C \le 0$ (K = 0.852..., C=-0.956...) otherwise, Atkinson switching algorithm is used; for $max(p, q) < 1$, method of Johnk is used; for $min(p, q) < 1$, $max(p, q) > 1$, Atkinson switching algorithm is used (CJA stands for the first letters of Cheng, Johnk, Atkinson); for $p = 1$ or $q = 1$, inverse cumulative distribution function method is used;for $p = 1$ and $q = 1$, beta distribution is reduced to uniform distribution.	Beta
BTPE	Acceptance/rejection method for $ntrial \cdot \min(p, 1 - p) \ge 30$ with decomposition into 4 regions:	Binomial

Method	Short Description	Functions
	– 2 parallelograms – triangle – left exponential tail – right exponential tail	
H2PE	Acceptance/rejection method for large mode of distribution with decomposition into 3 regions: – rectangular – left exponential tail – right exponential tail	Hypergeometric
PTPE	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into 4 regions: – 2 parallelograms – triangle – left exponential tail – right exponential tail; otherwise, table lookup method is used.	Poisson
POISNORM	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.	Poisson, PoissonV
NBAR	Acceptance/rejection method for , $\frac{(a-1)\cdot(1-p)}{p} \geq 100$ with decomposition into 5 regions: – rectangular – 2 trapezoid – left exponential tail – right exponential tail	NegBinomial
CHI2GAMMA	Random number generator of chi-square distribution with v degrees of freedom. To generate any successive random number x of the chi-square distribution: • If v is 1 or 3, a chi-square distributed random number is generated as a sum of squares of v independent normal random numbers with mean value a = 0 and standard deviation σ =1. • If v is even and $2 \leq v \leq 16$, a chi-square distributed random number is generated using the formula: $x = -2ln\left(\prod_{i=1}^{v/2} u_i\right)$ where u_i are successive random numbers uniformly distributed over the interval (0, 1) • If $v \geq 17$ or v is odd and $5 \leq v \leq 15$, a chi-square distribution is reduced to a Gamma distribution with these parameters:	ChiSquare

Method	Short Description	Functions
	• Shape $a = v / 2$ • Offset a = 0 • Scale factor $\beta = 2$ The random numbers of the Gamma distribution are generated using the VSL_RNG_METHOD_GAMMA_GNORM method.	

NOTE
In this document, routines are often referred to by their base name (`Gaussian`) when this does not lead to ambiguity. In the routine reference, the full name (`vsrnggaussian`, `vsRngGaussian`) is always used in prototypes and code examples.

Basic Generators

VS provides pseudorandom, quasi-random, and non-deterministic random number generators. This includes the following BRNGs, which differ in speed and other properties:

- the 31-bit multiplicative congruential pseudorandom number generator $MCG(1132489760, 2^{31} - 1)$ [L'Ecuyer99]
- the 32-bit generalized feedback shift register pseudorandom number generator $GFSR(250,103)$ [Kirkpatrick81]
- the combined multiple recursive pseudorandom number generator *MRG32k3a* [L'Ecuyer99a]
- the 59-bit multiplicative congruential pseudorandom number generator $MCG(13^{13}, 2^{59})$ from NAG Numerical Libraries [NAG]
- Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG]
- Mersenne Twister pseudorandom number generator *MT19937* [Matsumoto98] with period length 2^{19937}-1 of the produced sequence
- Set of 6024 Mersenne Twister pseudorandom number generators *MT2203* [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to 2^{2203}-1. Parameters of the generators provide mutual independence of the corresponding sequences.
- SIMD-oriented Fast Mersenne Twister pseudorandom number generator *SFMT19937* [Saito08] with a period length equal to 2^{19937}-1 of the produced sequence.
- Sobol quasi-random number generator [Sobol76], [Bratley88], which works in arbitrary dimension. For dimensions greater than 40 the user should supply initialization parameters (initial direction numbers and primitive polynomials or direction numbers) by using `vslNewStreamEx` function. See additional details on interface for registration of the parameters in the library in VS Notes.
- Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension. For dimensions greater than 318 the user should supply initialization parameters (irreducible polynomials or direction numbers) by using `vslNewStreamEx` function. See additional details on interface for registration of the parameters in the library in VS Notes.
- Non-deterministic random number generator (RDRAND-based generators only) [AVX], [IntelSWMan].

NOTE
You can use a non-deterministic random number generator only if the underlying hardware supports it. For instructions on how to detect if an Intel CPU supports a non-deterministic random number generator see, for example, *Chapter 8: Post-32nm Processor Instructions* in [AVX] or *Chapter 4: RdRand Instruction Usage* in [BMT].

NOTE
The time required by some non-deterministic sources to generate a random number is not constant, so you might have to make multiple requests before the next random number is available. VS limits the number of retries for requests to the non-deterministic source to 10. You can redefine the maximum number of retries during the initialization of the non-deterministic random number generator with the vslNewStreamEx function.

For more details on the non-deterministic source implementation for Intel CPUs please refer to Section 7.3.17, Volume 1, *Random Number Generator Instruction* in [IntelSWMan] and Section 4.2.2, *RdRand Retry Loop* in [BMT].

- Philox4x32-10 counter-based pseudorandom number generator with a period of 2^{128} *PHILOX4X32X10*[Salmon11].
- ARS-5 counter-based pseudorandom number generator with a period of 2^{128}, which uses instructions from the AES-NI set *ARS5*[Salmon11].

See some testing results for the generators in VS Notes and comparative performance data at https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html.

VS provides means of registration of such user-designed generators through the steps described in Advanced Service Routines.

For some basic generators, VS provides two methods of creating independent random streams in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo.

In addition, MT2203 pseudorandom number generator is a set of 6024 generators designed to create up to 6024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [Coddington94].

You may want to design and use your own basic generators. VS provides means of registration of such user-designed generators through the steps described in Advanced Service Routines.

There is also an option to utilize externally generated random numbers in VS distribution generator routines. For this purpose VS provides three additional basic random number generators:

- for external random data packed in 32-bit integer array
- for external random data stored in double precision floating-point array; data is supposed to be uniformly distributed over (a,b) interval
- for external random data stored in single precision floating-point array; data is supposed to be uniformly distributed over (a,b) interval.

Such basic generators are called the abstract basic random number generators.

See VS Notes for a more detailed description of the generator properties.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

BRNG Parameter Definition

Predefined values for the *brng* input parameter are as follows:

Values of *brng* parameter

Value	Short Description
VSL_BRNG_MCG31	A 31-bit multiplicative congruential generator.

Value	Short Description
VSL_BRNG_R250	A generalized feedback shift register generator.
VSL_BRNG_MRG32K3A	A combined multiple recursive generator with two components of order 3.
VSL_BRNG_MCG59	A 59-bit multiplicative congruential generator.
VSL_BRNG_WH	A set of 273 Wichmann-Hill combined multiplicative congruential generators.
VSL_BRNG_MT19937	A Mersenne Twister pseudorandom number generator.
VSL_BRNG_MT2203	A set of 6024 Mersenne Twister pseudorandom number generators.
VSL_BRNG_SFMT19937	A SIMD-oriented Fast Mersenne Twister pseudorandom number generator.
VSL_BRNG_SOBOL	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \le s \le 40$; user-defined dimensions are also available.
VSL_BRNG_NIEDERR	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \le s \le 318$; user-defined dimensions are also available.
VSL_BRNG_IABSTRACT	An abstract random number generator for integer arrays.
VSL_BRNG_DABSTRACT	An abstract random number generator for double precision floating-point arrays.
VSL_BRNG_SABSTRACT	An abstract random number generator for single precision floating-point arrays.
VSL_BRNG_NONDETERM	A non-deterministic random number generator.
VSL_BRNG_PHILOX4X32X10	A Philox4x32-10 counter-based pseudorandom number generator.
VSL_BRNG_ARS5	An ARS-5 counter-based pseudorandom number generator that uses instructions from the AES-NI set.

See VS Notes for detailed description.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Random Streams

Random stream (or *stream*) is an abstract source of pseudo- and quasi-random sequences of uniform distribution. You can operate with stream state descriptors only. A stream state descriptor, which holds state descriptive information for a particular BRNG, is a necessary parameter in each routine of a distribution generator. Only the distribution generator routines operate with random streams directly. See VS Notes for details.

NOTE
Random streams associated with abstract basic random number generator are called the abstract random streams. See VS Notes for detailed description of abstract streams and their use.

You can create unlimited number of random streams by VS Service Routines like `NewStream` and utilize them in any distribution generator to get the sequence of numbers of given probability distribution. When they are no longer needed, the streams should be deleted calling service routine `DeleteStream`.

VS provides service functions `SaveStreamF` and `LoadStreamF` to save random stream descriptive data to a binary file and to read this data from a binary file respectively. See VS Notes for detailed description.

BRNG Data Types

FORTRAN 77:

```
INTEGER*4 vslstreamstate(2)
```

Fortran 90:

```
TYPEVSL_STREAM_STATEINTEGER*4 descriptor1INTEGER*4 descriptor2ENDTYPE VSL_STREAM_STATE
```

Error Reporting

VS RNG routines return status codes of the performed operation to report errors to the calling program. The application should perform error-related actions and/or recover from the error. The status codes are of integer type and have the following format:

`VSL_ERROR_<`*ERROR_NAME*> - indicates VS errors common for all VS domains.

`VSL_RNG_ERROR_<`*ERROR_NAME*> - indicates VS RNG errors.

VS RNG errors are of negative values while warnings are of positive values. The status code of zero value indicates successful completion of the operation: `VSL_ERROR_OK` (or synonymic `VSL_STATUS_OK`).

Status Codes

Status Code	Description
Common VSL	
`VSL_ERROR_OK`, `VSL_STATUS_OK`	No error, execution is successful.
`VSL_ERROR_BADARGS`	Input argument value is not valid.
`VSL_ERROR_CPU_NOT_SUPPORTED`	CPU version is not supported.
`VSL_ERROR_FEATURE_NOT_IMPLEMENTED`	Feature invoked is not implemented.
`VSL_ERROR_MEM_FAILURE`	System cannot allocate memory.
`VSL_ERROR_NULL_PTR`	Input pointer argument is NULL.
`VSL_ERROR_UNKNOWN`	Unknown error.
VS RNG Specific	
`VSL_RNG_ERROR_BAD_FILE_FORMAT`	File format is unknown.
`VSL_RNG_ERROR_BAD_MEM_FORMAT`	Descriptive random stream format is unknown.
`VSL_RNG_ERROR_BAD_NBITS`	The value in `NBits` field is bad.

Status Code	Description
VSL_RNG_ERROR_BAD_NSEEDS	The value in NSeeds field is bad.
VSL_RNG_ERROR_BAD_STREAM	The random stream is invalid.
VSL_RNG_ERROR_BAD_STREAM_STATE_SIZE	The value in StreamStateSize field is bad.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or >nmax.
VSL_RNG_ERROR_BAD_WORD_SIZE	The value in WordSize field is bad.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_BRNG_TABLE_FULL	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.
VSL_RNG_ERROR_BRNGS_INCOMPATIBLE	Two BRNGs are not compatible for the operation.
VSL_RNG_ERROR_FILE_CLOSE	Error in closing the file.
VSL_RNG_ERROR_FILE_OPEN	Error in opening the file.
VSL_RNG_ERROR_FILE_READ	Error in reading the file.
VSL_RNG_ERROR_FILE_WRITE	Error in writing the file.
VSL_RNG_ERROR_INVALID_ABSTRACT_STREAM	The abstract random stream is invalid.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is not valid.
VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns zero as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator is exceeded.
VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support Skip-Ahead method.
VSL_RNG_ERROR_SKIPAHEADEX_UNSUPPORTED	BRNG does not support advanced Skip-Ahead method.
VSL_RNG_ERROR_UNSUPPORTED_FILE_VER	File format version is not supported.
VSL_RNG_ERROR_NONDETERM_NOT_SUPPORTED	Non-deterministic random number generator is not supported on the CPU running the application.
VSL_RNG_ERROR_NONDETERM_ NRETRIES_EXCEEDED	Number of retries to generate a random number using non-deterministic random number generator exceeds threshold (see Section 7.2.1.12 *Non-deterministic* in [VS Notes] for more details)
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

VS RNG Usage ModelIntel® oneMKL RNG Usage Model

A typical algorithm for VSoneMKL random number generators is as follows:

1. Create and initialize stream/streams. Functions `vslNewStream`, `vslNewStreamEx`, `vslCopyStream`, `vslCopyStreamState`, `vslLeapfrogStream`, `vslSkipAheadStream`, `vslSkipAheadStreamEx`.
2. Call one or more RNGs.
3. Process the output.
4. Delete the stream or streams with the function `vslDeleteStream`.

NOTE
You may reiterate steps 2-3. Random number streams may be generated for different threads.

The following example demonstrates generation of a random stream that is output of basic generator MT19937. The seed is equal to 777. The stream is used to generate 10,000 normally distributed random numbers in blocks of 1,000 random numbers with parameters *a* = 5 and *sigma* = 2. Delete the streams after completing the generation. The purpose of the example is to calculate the sample mean for normal distribution with the given parameters.

Example of VS RNG Usage

```
      include 'mkl_vsl.f90'

      program MKL_VSL_GAUSSIAN

      USE MKL_VSL_TYPE
      USE MKL_VSL

      real(kind=8) r(1000)  ! buffer for random numbers
      real(kind=8) s        ! average
      real(kind=8) a, sigma ! parameters of normal distribution

      TYPE (VSL_STREAM_STATE) :: stream

      integer(kind=4) errcode
      integer(kind=4) i,j
      integer brng,method,seed,n

      n = 1000
      s = 0.0
      a = 5.0
      sigma  = 2.0
      brng=VSL_BRNG_MT19937
      method=VSL_RNG_METHOD_GAUSSIAN_ICDF
      seed=777

!     ***** Initializing *****
      errcode=vslnewstream( stream, brng,  seed )

!     ***** Generating *****
      do i = 1,10
```

```
          errcode=vdrnggaussian( method, stream, n, r, a, sigma )
          do j = 1, 1000
              s = s + r(j)
          end do
      end do

      s = s / 10000.0

!     ***** Deinitialize *****
      errcode=vsldeletestream( stream )

!     ***** Printing results *****
      print *,"Sample mean of normal distribution = ", s

      end
```

Additionally, examples that demonstrate usage of VS random number generators are available in:

`${MKL}/examples/vslf/source`

Service Routines

Stream handling comprises routines for creating, deleting, or copying the streams and getting the index of a basic generator. A random stream can also be saved to and then read from a binary file. Table "Service Routines" lists all available service routines

Service Routines

Routine	Short Description
vslNewStream	Creates and initializes a random stream.
vslNewStreamEx	Creates and initializes a random stream for the generators with multiple initial conditions.
vsliNewAbstractStream	Creates and initializes an abstract random stream for integer arrays.
vsldNewAbstractStream	Creates and initializes an abstract random stream for double precision floating-point arrays.
vslsNewAbstractStream	Creates and initializes an abstract random stream for single precision floating-point arrays.
vslDeleteStream	Deletes previously created stream.
vslCopyStream	Copies a stream to another stream.
vslCopyStreamState	Creates a copy of a random stream state.
vslSaveStreamF	Writes a stream to a binary file.
vslLoadStreamF	Reads a stream from a binary file.
vslSaveStreamM	Writes a random stream descriptive data, including state, to a memory buffer.
vslLoadStreamM	Creates a new stream and reads stream descriptive data, including state, from the memory buffer.
vslGetStreamSize	Computes size of memory necessary to hold the random stream.

Routine	Short Description
vslLeapfrogStream	Initializes the stream by the leapfrog method to generate a subsequence of the original sequence.
vslSkipAheadStream	Initializes the stream by the skip-ahead method.
vslSkipAheadStreamEx	Initializes the stream by the advanced skip-ahead method.
vslGetStreamStateBrng	Obtains the index of the basic generator responsible for the generation of a given random stream.
vslGetNumRegBrngs	Obtains the number of currently registered basic generators.

Most of the generator-based work comprises three basic steps:

1. Creating and initializing a stream (vslNewStream, vslNewStreamEx, vslCopyStream, vslCopyStreamState, vslLeapfrogStream, vslSkipAheadStream, vslSkipAheadStreamEx).
2. Generating random numbers with given distribution, see Distribution Generators.
3. Deleting the stream (vslDeleteStream).

Note that you can concurrently create multiple streams and obtain random data from one or several generators by using the stream state. You must use the vslDeleteStream function to delete all the streams afterwards.

vslNewStream

Creates and initializes a random stream.

Syntax

```
status = vslnewstream( stream, brng, seed )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
brng	INTEGER, INTENT(IN)	Index of the basic generator to initialize the stream. See Table Values of *brng* parameter for specific value.
seed	INTEGER, INTENT(IN)	Initial condition of the stream. In the case of a quasi-random number generator seed parameter is used to set the dimension. If the dimension is greater than the dimension that brng can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Stream state descriptor

Description

For a basic generator with number *brng*, this function creates a new stream and initializes it with a 32-bit seed. The seed is an initial value used to select a particular sequence generated by the basic generator *brng*. The function is also applicable for generators with multiple initial conditions. Use this function to create and

initialize a new stream with a 32-bit seed only. If you need to provide multiple initial conditions such as several 32-bit or wider seeds, use the function vslNewStreamEx. See VS Notes for a more detailed description of stream initialization for different basic generators.

NOTE
This function is not applicable for abstract basic random number generators. Please use `vsliNewAbstractStream`, `vslsNewAbstractStream` or `vsldNewAbstractStream` to utilize integer, single-precision or double-precision external random data respectively.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_RNG_ERROR_INVALID_BRNG_INDEX`	BRNG index is invalid.
`VSL_ERROR_MEM_FAILURE`	System cannot allocate memory for *stream*.
`VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED`	Non-deterministic random number generator is not supported.
`VSL_RNG_ERROR_ARS5_NOT_SUPPORTED`	ARS-5 random number generator is not supported on the CPU running the application.

vslNewStreamEx

Creates and initializes a random stream for generators with multiple initial conditions.

Syntax

```
status = vslnewstreamex( stream, brng, n, params )
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
brng	`INTEGER`, `INTENT(IN)`	Index of the basic generator to initialize the stream. See Table "Values of *brng* parameter" for specific value.
n	`INTEGER`, `INTENT(IN)`	Number of initial conditions contained in *params*
params	`INTEGER(KIND=4)`, `INTENT(IN)`	Array of initial conditions necessary for the basic generator *brng* to initialize the stream. In the case of a quasi-random number generator only the first element in *params*

Name	Type	Description
		parameter is used to set the dimension. If the dimension is greater than the dimension that *brng* can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Stream state descriptor

Description

The vslNewStreamEx function provides an advanced tool to set the initial conditions for a basic generator if its input arguments imply several initialization parameters. Initial values are used to select a particular sequence generated by the basic generator *brng*. Whenever possible, use vslNewStream, which is analogous to vslNewStreamEx except that it takes only one 32-bit initial condition. In particular, vslNewStreamEx may be used to initialize the state table in Generalized Feedback Shift Register Generators (GFSRs). A more detailed description of this issue can be found in VS Notes.

This function is also used to pass user-defined initialization parameters of quasi-random number generators into the library. See VS Notes for the format for their passing and registration in VS.

NOTE
This function is not applicable for abstract basic random number generators. Please use vsliNewAbstractStream, vslsNewAbstractStream or vsldNewAbstractStream to utilize integer, single-precision or double-precision external random data respectively.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *stream*.
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vsliNewAbstractStream

Creates and initializes an abstract random stream for integer arrays.

Syntax

status = vslinewabstractstream(*stream*, *n*, *ibuf*, *icallback*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Size of the array *ibuf*
ibuf	INTEGER(KIND=4), INTENT(IN)	Array of *n* 32-bit integers
icallback	See *Note* below	Address of the callback function used for *ibuf* update

NOTE
Format of the callback function in FORTRAN 77:

```
INTEGER FUNCTION IUPDATEFUNC( stream, n, ibuf, nmin, nmax, idx )
INTEGER*4 stream(2)
INTEGER n
INTEGER*4 ibuf(n)
INTEGER nmin
INTEGER nmax
INTEGER idx
```

Format of the callback function in Fortran 90:

```
INTEGER FUNCTION IUPDATEFUNC[C]( stream, n, ibuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE),POINTER :: stream[reference]
INTEGER(KIND=4),INTENT(IN)     :: n[reference]
INTEGER(KIND=4),INTENT(OUT)    :: ibuf[reference](0:n-1)
INTEGER(KIND=4),INTENT(IN)     :: nmin[reference]
INTEGER(KIND=4),INTENT(IN)     :: nmax[reference]
INTEGER(KIND=4),INTENT(IN)     :: idx[reference]
```

The callback function returns the number of elements in the array actually updated by the function.Table `icallback` Callback Function Parameters gives the description of the callback function parameters.

icallback Callback Function Parameters

Parameters	Short Description
stream	Abstract random stream descriptor
n	Size of *ibuf*
ibuf	Array of random numbers associated with the stream *stream*

Parameters	Short Description
nmin	Minimal quantity of numbers to update
nmax	Maximal quantity of numbers that can be updated
idx	Position in cyclic buffer *ibuf* to start update 0≤*idx*<*n*.

Output Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), TINTENT(OUT)	Descriptor of the stream state structure

Description

The vsliNewAbstractStream function creates a new abstract stream and associates it with an integer array *ibuf* and your callback function *icallback* that is intended for updating of *ibuf* content.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BADARGS	Parameter *n* is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *stream*.
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

vsldNewAbstractStream

Creates and initializes an abstract random stream for double precision floating-point arrays.

Syntax

status = vsldnewabstractstream(*stream*, *n*, *dbuf*, *a*, *b*, *dcallback*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Size of the array *dbuf*
dbuf	REAL(KIND=8), INTENT(IN)	Array of *n* double precision floating-point random numbers with uniform distribution over interval (a,b)
a	REAL(KIND=8), INTENT(IN)	Left boundary a
b	REAL(KIND=8), INTENT(IN)	Right boundary b
dcallback	See *Note* below	Address of the callback function used for update of the array *dbuf*

Output Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Descriptor of the stream state structure

NOTE
Format of the callback function in FORTRAN 77:

```
INTEGER FUNCTION DUPDATEFUNC( stream, n, dbuf, nmin, nmax, idx )
INTEGER*4 stream(2)
INTEGER n
DOUBLE PRECISION dbuf(n)
INTEGER nmin
INTEGER nmax
INTEGER idx
```

Format of the callback function in Fortran 90:

```
INTEGER FUNCTION DUPDATEFUNC[C]( stream, n, dbuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE),POINTER   :: stream[reference]
INTEGER(KIND=4),INTENT(IN)       :: n[reference]
REAL(KIND=8),INTENT(OUT)         :: dbuf[reference](0:n-1)
INTEGER(KIND=4),INTENT(IN)       :: nmin[reference]
INTEGER(KIND=4),INTENT(IN)       :: nmax[reference]
INTEGER(KIND=4),INTENT(IN)       :: idx[reference]
```

The callback function returns the number of elements in the array actually updated by the function.Table `dcallback` Callback Function Parameters gives the description of the callback function parameters.

dcallback Callback Function Parameters

Parameters	Short Description
stream	Abstract random stream descriptor
n	Size of *dbuf*
dbuf	Array of random numbers associated with the stream *stream*
nmin	Minimal quantity of numbers to update
nmax	Maximal quantity of numbers that can be updated
idx	Position in cyclic buffer *dbuf* to start update $0 \le idx < n$.

Description

The `vsldNewAbstractStream` function creates a new abstract stream for double precision floating-point arrays with random numbers of the uniform distribution over interval (`a`,`b`). The function associates the stream with a double precision array *dbuf* and your callback function *dcallback* that is intended for updating of *dbuf* content.

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.

VSL_ERROR_BADARGS	Parameter *n* is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *stream*.
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

vslsNewAbstractStream

Creates and initializes an abstract random stream for single precision floating-point arrays.

Syntax

status = vslsnewabstractstream(*stream*, *n*, *sbuf*, *a*, *b*, *scallback*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT(IN)	Size of the array *sbuf*
sbuf	REAL(KIND=4), INTENT(IN)	Array of *n* single precision floating-point random numbers with uniform distribution over interval (a,b)
a	REAL(KIND=4), INTENT(IN)	Left boundary a
b	REAL(KIND=4), INTENT(IN)	Right boundary b
scallback	See *Note* below	Address of the callback function used for update of the array *sbuf*

Output Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Descriptor of the stream state structure

NOTE
Format of the callback function in FORTRAN 77:

```
INTEGER FUNCTION SUPDATEFUNC( stream, n, ibuf, nmin, nmax, idx )
INTEGER*4 stream(2)
INTEGER n
REAL sbuf(n)
INTEGER nmin
INTEGER nmax
INTEGER idx
```

Format of the callback function in Fortran 90:

```
INTEGER FUNCTION SUPDATEFUNC[C]( stream, n, sbuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE),POINTER :: stream[reference]
INTEGER(KIND=4),INTENT(IN)     :: n[reference]
REAL(KIND=4),   INTENT(OUT)    :: sbuf[reference](0:n-1)
INTEGER(KIND=4),INTENT(IN)     :: nmin[reference]
INTEGER(KIND=4),INTENT(IN)     :: nmax[reference]
INTEGER(KIND=4),INTENT(IN)     :: idx[reference]
```

The callback function returns the number of elements in the array actually updated by the function.Table `scallback` Callback Function Parameters gives the description of the callback function parameters.

scallback Callback Function Parameters

Parameters	Short Description
stream	Abstract random stream descriptor
n	Size of *sbuf*
sbuf	Array of random numbers associated with the stream *stream*
nmin	Minimal quantity of numbers to update
nmax	Maximal quantity of numbers that can be updated
idx	Position in cyclic buffer *sbuf* to start update $0 \le idx < n$.

Description

The `vslsNewAbstractStream` function creates a new abstract stream for single precision floating-point arrays with random numbers of the uniform distribution over interval (`a`,`b`). The function associates the stream with a single precision array *sbuf* and your callback function *scallback* that is intended for updating of *sbuf* content.

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_BADARGS`	Parameter *n* is not positive.
`VSL_ERROR_MEM_FAILURE`	System cannot allocate memory for *stream*.
`VSL_ERROR_NULL_PTR`	Either buffer or callback function parameter is a `NULL` pointer.

vslDeleteStream

Deletes a random stream.

Syntax

```
status = vsldeletestream( stream )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input/Output Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the descriptor becomes invalid.

Description

The function deletes the random stream created by one of the initialization functions.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* parameter is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.

vslCopyStream

Creates a copy of a random stream.

Syntax

```
status = vslcopystream( newstream, srcstream )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
srcstream	TYPE(VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream to be copied

Output Parameters

Name	Type	Description
newstream	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Copied random stream descriptor

Description

The function creates an exact copy of *srcstream* and stores its descriptor to *newstream*.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*srcstream* parameter is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*srcstream* is not a valid random stream.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *newstream*.

vslCopyStreamState

Creates a copy of a random stream state.

Syntax

status = vslcopystreamstate(*deststream*, *srcstream*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
srcstream	TYPE(VSL_STREAM_STATE), INTENT(IN)	Descriptor of the destination stream where the state of *scrstream* stream is copied

Output Parameters

Name	Type	Description
deststream	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Descriptor of the stream with the state to be copied

Description

The vslCopyStreamState function copies a stream state from *srcstream* to the existing *deststream* stream. Both the streams should be generated by the same basic generator. An error message is generated when the index of the BRNG that produced *deststream* stream differs from the index of the BRNG that generated *srcstream* stream.

Unlike vslCopyStream function, which creates a new stream and copies both the stream state and other data from *srcstream*, the function vslCopyStreamState copies only *srcstream* stream state data to the generated *deststream* stream.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either *srcstream* or *deststream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	Either *srcstream* or *deststream* is not a valid random stream.
VSL_RNG_ERROR_BRNGS_INCOMPATIBLE	BRNG associated with *srcstream* is not compatible with BRNG associated with *deststream*.

vslSaveStreamF

Writes random stream descriptive data, including stream state, to binary file.

Syntax

```
errstatus = vslsavestreamf( stream, fname )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(IN)	Random stream to be written to the file
fname	CHARACTER(*), INTENT(IN)	File name specified as a C-style null-terminated string

Output Parameters

Name	Type	Description
errstatus	INTEGER	Error status of the operation

Description

The vslSaveStreamF function writes the random stream descriptive data, including the stream state, to the binary file. Random stream descriptive data is saved to the binary file with the name *fname*. The random stream *stream* must be a valid stream created by vslNewStream-like or vslCopyStream-like service routines. If the stream cannot be saved to the file, *errstatus* has a non-zero value. The random stream can be read from the binary file using the vslLoadStreamF function.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either *fname* or *stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_RNG_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_RNG_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.

vslLoadStreamF

Creates new stream and reads stream descriptive data, including stream state, from binary file.

Syntax

```
errstatus = vslloadstreamf( stream, fname )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
fname	CHARACTER(*), INTENT(IN)	File name specified as a C-style null-terminated string

Output Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Descriptor of a new random stream
errstatus	INTEGER	Error status of the operation

Description

The vslLoadStreamF function creates a new stream and reads stream descriptive data, including the stream state, from the binary file. A new random stream is created using the stream descriptive data from the binary file with the name *fname*. If the stream cannot be read (for example, an I/O error occurs or the file format is invalid), *errstatus* has a non-zero value. To save random stream to the file, use vslSaveStreamF function.

Caution
Calling vslLoadStreamF with a previously initialized *stream* pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling vslLoadStreamF, you should call the vslDeleteStream function first to deallocate the resources.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*fname* is a NULL pointer.
VSL_RNG_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_RNG_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_RNG_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.
VSL_RNG_ERROR_BAD_FILE_FORMAT	Unknown file format.
VSL_RNG_ERROR_UNSUPPORTED_FILE_VER	File format version is unsupported.

VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vslSaveStreamM

Writes random stream descriptive data, including stream state, to a memory buffer.

Syntax

```
errstatus = vslsavestreamm( stream, memptr )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(IN)	Random stream to be written to the memory
memptr	INTEGER(KIND=1), DIMENSION(*), INTENT(IN)	Memory buffer to save random stream descriptive data to

Output Parameters

Name	Type	Description
errstatus	INTEGER	Error status of the operation

Description

The vslSaveStreamM function writes the random stream descriptive data, including the stream state, to the memory at *memptr*. Random stream *stream* must be a valid stream created by vslNewStream-like or vslCopyStream-like service routines. The *memptr* parameter must be a valid pointer to the memory of size sufficient to hold the random stream *stream*. Use the service routine vslGetStreamSize to determine this amount of memory.

If the stream cannot be saved to the memory, *errstatus* has a non-zero value. The random stream can be read from the memory pointed by *memptr* using the vslLoadStreamM function.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either *memptr* or *stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is a NULL pointer.

vslLoadStreamM

Creates a new stream and reads stream descriptive data, including stream state, from the memory buffer.

Syntax

```
errstatus = vslloadstreamm( stream, memptr )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
memptr	INTEGER(KIND=1), DIMENSION(*), INTENT(IN)	Memory buffer to load random stream descriptive data from

Output Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Descriptor of a new random stream
errstatus	INTEGER	Error status of the operation

Description

The vslLoadStreamM function creates a new stream and reads stream descriptive data, including the stream state, from the memory buffer. A new random stream is created using the stream descriptive data from the memory pointer by *memptr*. If the stream cannot be read (for example, *memptr* is invalid), *errstatus* has a non-zero value. To save random stream to the memory, use vslSaveStreamM function. Use the service routine vslGetStreamSize to determine the amount of memory sufficient to hold the random stream.

Caution
Calling LoadStreamM with a previously initialized *stream* pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling vslLoadStreamM, you should call the vslDeleteStream function first to deallocate the resources.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*memptr* is a NULL pointer.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.
VSL_RNG_ERROR_BAD_MEM_FORMAT	Descriptive random stream format is unknown.

VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vslGetStreamSize

Computes size of memory necessary to hold the random stream.

Syntax

```
memsize = vslgetstreamsize( stream )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(IN)	Random stream

Output Parameters

Name	Type	Description
memsize	INTEGER	Amount of memory in bytes necessary to hold descriptive data of random stream *stream*

Description

The vslGetStreamSize function returns the size of memory in bytes which is necessary to hold the given random stream. Use the output of the function to allocate the buffer to which you will save the random stream by means of the vslSaveStreamM function.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_BAD_STREAM	*stream* is a NULL pointer.

vslLeapfrogStream

Initializes a stream using the leapfrog method.

Syntax

```
status = vslleapfrogstream( stream, k, nstreams )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream to which leapfrog method is applied
k	INTEGER, INTENT(IN)	Index of the computational node, or stream number
nstreams	INTEGER, INTENT(IN)	Largest number of computational nodes, or stride

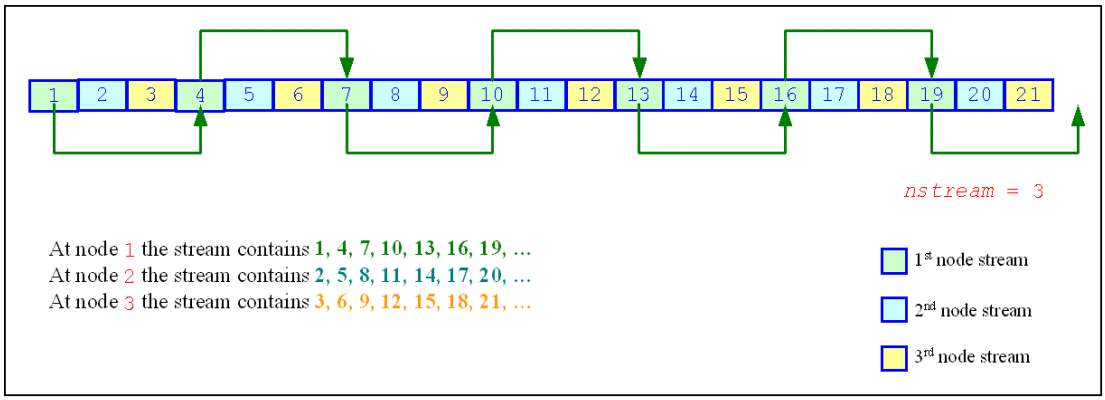
Description

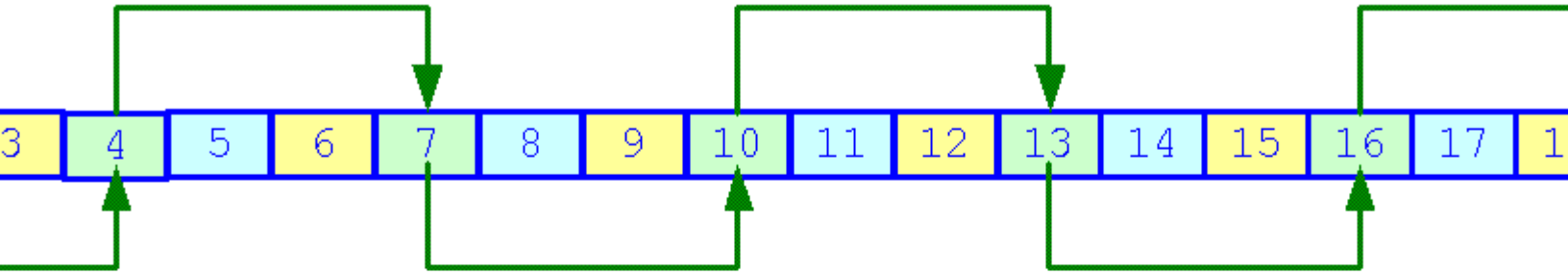
The vslLeapfrogStream function generates random numbers in a random stream with non-unit stride. This feature is particularly useful in distributing random numbers from the original stream across the *nstreams* buffers without generating the original random sequence with subsequent manual distribution.

One of the important applications of the leapfrog method is splitting the original sequence into non-overlapping subsequences across *nstreams* computational nodes. The function initializes the original random stream (see Figure "Leapfrog Method") to generate random numbers for the computational node k, $0 \le k < nstreams$, where *nstreams* is the largest number of computational nodes used.

__border__top

Leapfrog Method





he stream contains 1, 4, 7, 10, 13, 16, 19, ...
he stream contains 2, 5, 8, 11, 14, 17, 20, ...
he stream contains 3, 6, 9, 12, 15, 18, 21, ...

The leapfrog method is supported only for those basic generators that allow splitting elements by the leapfrog method, which is more efficient than simply generating them by a generator with subsequent manual distribution across computational nodes. See VS Notes for details.

For quasi-random basic generators, the leapfrog method allows generating individual components of quasi-random vectors instead of whole quasi-random vectors. In this case *nstreams* parameter should be equal to the dimension of the quasi-random vector while *k* parameter should be the index of a component to be generated (0 ≤*k* < *nstreams*). Other parameters values are not allowed.

The following code illustrates the initialization of three independent streams using the leapfrog method:

Code for Leapfrog Method

```
...
TYPE(VSL_STREAM_STATE)   ::stream1
TYPE(VSL_STREAM_STATE)   ::stream2
TYPE(VSL_STREAM_STATE)   ::stream3

! Creating 3 identical streams
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)
status = vslcopystream(stream2, stream1)
status = vslcopystream(stream3, stream1)

! Leapfrogging the streams
status = vslleapfrogstream(stream1, 0, 3)
status = vslleapfrogstream(stream2, 1, 3)
status = vslleapfrogstream(stream3, 2, 3)

! Generating random numbers
...
! Deleting the streams
status = vsldeletestream(stream1)
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...
```

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.

vslSkipAheadStream

Initializes a stream using the block-splitting method.

Syntax

status = vslskipaheadstream(*stream*, *nskip*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

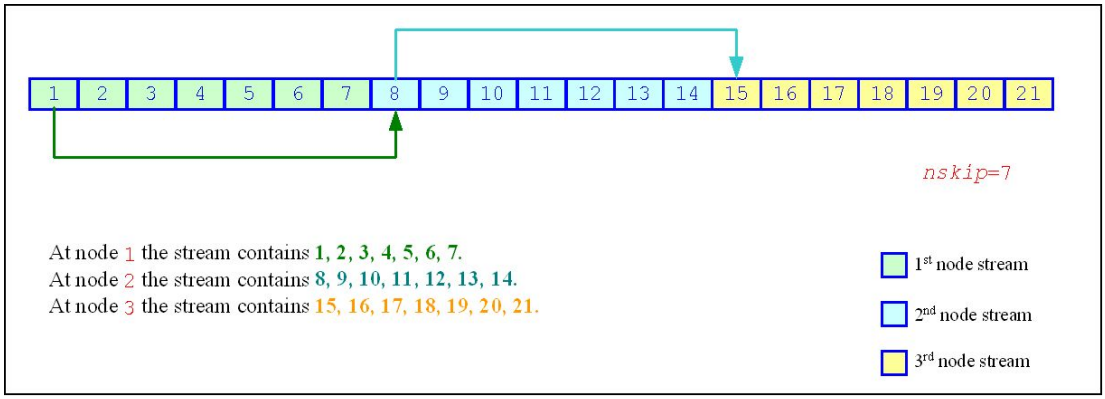
Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream to which block-splitting method is applied
nskip	INTEGER(KIND=8), INTENT(IN)	Number of skipped elements

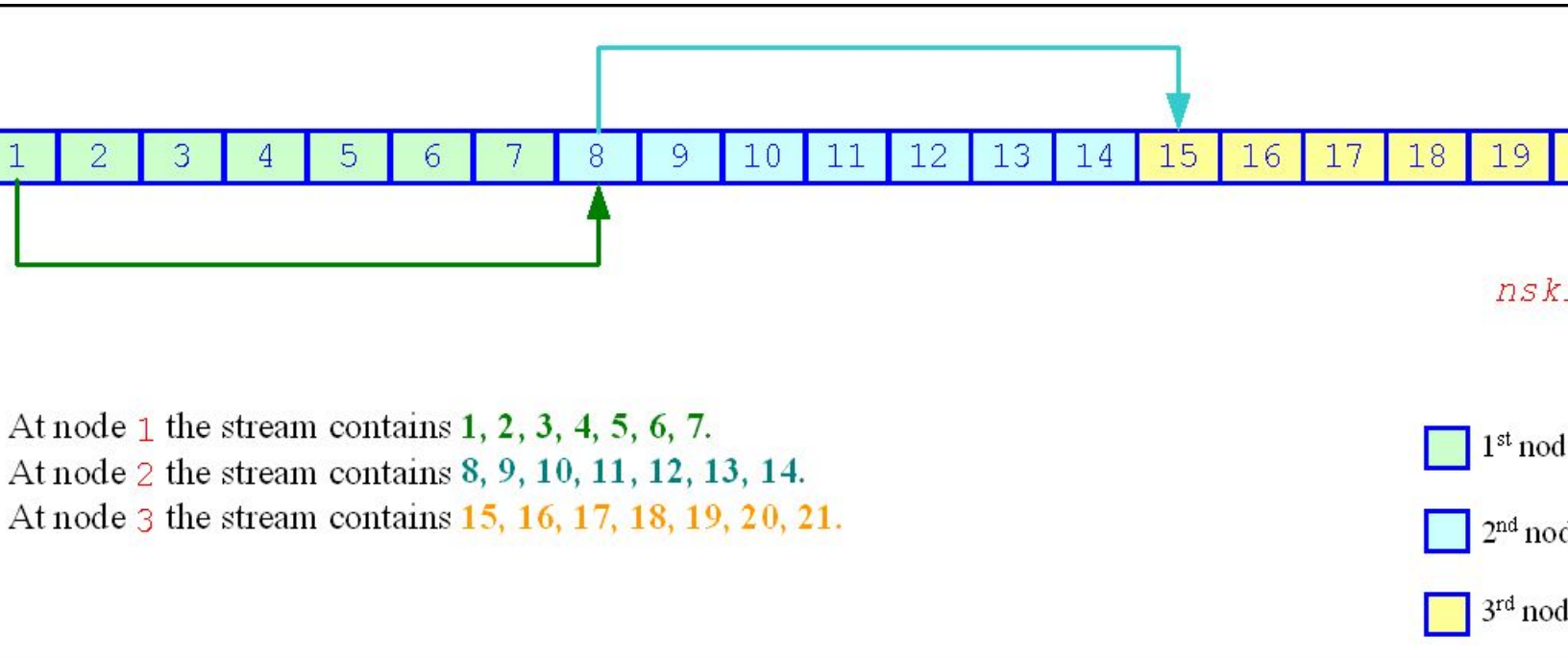
Description

The `vslSkipAheadStream` function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is *nskip*, then the original random sequence may be split by `vslSkipAheadStream` into non-overlapping blocks of *nskip* size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. (see Figure "Block-Splitting Method").

__border__top

Block-Splitting Method





The skip-ahead method is supported only for those basic generators that allow skipping elements by the skip-ahead method, which is more efficient than simply generating them by generator with subsequent manual skipping. See VS Notes for details.

Please note that for quasi-random basic generators the skip-ahead method works with components of quasi-random vectors rather than with whole quasi-random vectors. Therefore, to skip NS quasi-random vectors, set the *nskip* parameter equal to the NS*DIMEN, where DIMEN is the dimension of the quasi-random vector. If this operation results in exceeding the period of the quasi-random number generator, which is 2^{32}-1, the library returns the VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED error code.

The following code illustrates how to initialize three independent streams using the vslSkipAheadStream function:

Code for Block-Splitting Method

```
...
type(VSL_STREAM_STATE)  ::stream1
type(VSL_STREAM_STATE)  ::stream2
type(VSL_STREAM_STATE)  ::stream3

! Creating the 1st stream
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)

! Skipping ahead by 7 elements the 2nd stream
status = vslcopystream(stream2, stream1);
status = vslskipaheadstream(stream2, 7);

! Skipping ahead by 7 elements the 3rd stream
status = vslcopystream(stream3, stream2);
status = vslskipaheadstream(stream3, 7);

! Generating random numbers
...
! Deleting the streams
status = vsldeletestream(stream1)
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...
```

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support the Skip-Ahead method.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the quasi-random number generator is exceeded.

vslSkipAheadStreamEx

Initializes a stream using the block-splitting method with partitioned number of skipped elements.

Syntax

status = vslskipAheadstreamEx(*stream*, *n*, *nskip*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), INTENT(IN)	Pointer to the stream state structure to which block-splitting method is applied
n	INTEGER, INTENT(IN)	Number of summands in *nskip*
nskip	INTEGER(KIND=8), INTENT(IN)	Partitioned number of skipped elements

Description

The vslSkipAheadStreamEx function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is *nskip*, then the original random sequence may be split by vslSkipAheadStreamEx into non-overlapping blocks of *nskip* size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method.

Use this function when the number of elements to skip in a random stream is greater than 2^{63}. Prior calls to the function represent the number of skipped elements with array of size *n* as shown below:

nskip[0]+ *nskip*[1]*2^{64}+*nskip*[2]* 2^{128}+ … +*nskip*[n-1]*$2^{(64*(n-1))}$;

When the number of skipped elements is less than 2^{63} you can use either vslSkipAheadtreamEx or vslSkipAheadStream. The following code illustrates how to initialize three independent streams using the vslSkipAheadStreamEx function:

```
...
type(VSL_STREAM_STATE) ::stream1 type(VSL_STREAM_STATE) ::stream2
type(VSL_STREAM_STATE) ::stream3

! Creating the 1st stream
status = vslnewstream(stream1, VSL_BRNG_MRG32K3A, 174)

! To skip 2^64 elements in the random stream skipaheadstreamEx(nskip) function should !    be
called with nskip represented as nskip = 2^64 = 0 + 1 * 2^64
integer(kind=8) nskip(2)
nskip(1) = 0
nskip(2) = 1

! Skipping ahead by 2^64 elements the 2nd stream
status = vslcopystream(stream2, stream1)
status = vslskipaheadstreamex (stream2, 2, nskip)

! Skipping ahead by 2^64 elements the 3rd stream
status = vslcopystream(stream3, stream2)
status = vslskipaheadstreamex (stream3, 2, nskip)

! Generating random numbers
...

! Deleting the streams
status = vsldeletestream(stream1)
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...
```

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_SKIPAHEADEX_UNSUPPORTED	BRNG does not support the advanced Skip-Ahead method.

vslGetStreamStateBrng

Returns index of a basic generator used for generation of a given random stream.

Syntax

```
brng = vslgetstreamstatebrng( stream )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
stream	TYPE(VSL_STREAM_STATE), TINTENT(IN)	Descriptor of the stream state

Output Parameters

Name	Type	Description
brng	INTEGER	Index of the basic generator assigned for the generation of *stream* ; negative in case of an error

Description

The vslGetStreamStateBrng function retrieves the index of a basic generator used for generation of a given random stream.

Return Values

VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.

vslGetNumRegBrngs

Obtains the number of currently registered basic generators.

Syntax

```
nregbrngs = vslgetnumregbrngs( )
```

Include Files

- mkl.fi, mkl_vsl.f90

Output Parameters

Name	Type	Description
nregbrngs	INTEGER	Number of basic generators registered at the moment of the function call

Description

The `vslGetNumRegBrngs` function obtains the number of currently registered basic generators. Whenever user registers a user-designed basic generator, the number of registered basic generators is incremented. The maximum number of basic generators that can be registered is determined by the `VSL_MAX_REG_BRNGS` parameter.

Distribution Generators

oneMKLVS routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence and the explanation of input and output parameters. Table "Continuous Distribution Generators" and Table "Discrete Distribution Generators" list the random number generator routines with data types and output distributions, and sets correspondence between data types of the generator routines and the basic random number generators.

Continuous Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
vRngUniform	s, d	s, d	Uniform continuous distribution on the interval [*a*,*b*)
vRngGaussian	s, d	s, d	Normal (Gaussian) distribution
vRngGaussianMV	s, d	s, d	Normal (Gaussian) multivariate distribution
vRngExponential	s, d	s, d	Exponential distribution
vRngLaplace	s, d	s, d	Laplace distribution (double exponential distribution)
vRngWeibull	s, d	s, d	Weibull distribution
vRngCauchy	s, d	s, d	Cauchy distribution
vRngRayleigh	s, d	s, d	Rayleigh distribution
vRngLognormal	s, d	s, d	Lognormal distribution
vRngGumbel	s, d	s, d	Gumbel (extreme value) distribution
vRngGamma	s, d	s, d	Gamma distribution
vRngBeta	s, d	s, d	Beta distribution
vRngChiSquare	s, d	s, d	Chi-Square distribution

Discrete Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
vRngUniform	i	d	Uniform discrete distribution on the interval [*a*,*b*)
vRngUniformBits	i	i	Underlying BRNG integer recurrence
vRngUniformBits32	i	i	Uniformly distributed bits in 32-bit chunks
vRngUniformBits64	i	i	Uniformly distributed bits in 64-bit chunks
vRngBernoulli	i	s	Bernoulli distribution
vRngGeometric	i	s	Geometric distribution
vRngBinomial	i	d	Binomial distribution
vRngHypergeometric	i	d	Hypergeometric distribution
vRngPoisson	i	s (for VSL_RNG_METHOD_POISSON_POISNORM) s (for distribution parameter $\lambda \geq 27$) and d (for $\lambda < 27$) (for VSL_RNG_METHOD_POISSON_PTPE)	Poisson distribution
vRngPoisson	i	s	Poisson distribution with varying mean
vRngNegBinomial	i	d	Negative binomial distribution, or Pascal distribution
vRngMultinomial	i	d	Multinomial distribution

Modes of random number generation

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for the applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval [*a*,*b*] belong to this interval irrespective of what *a* and *b* values may be. Fast mode provides high performance of generation and also guarantees that generated random numbers belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying relevant value of the method parameter in generator routines. List of distributions that support accurate mode of generation is given in the table below.

Distribution Generators Supporting Accurate Mode

Type of Distribution	Data Types
vRngUniform	s, d

Type of Distribution	Data Types
vRngExponential	s, d
vRngWeibull	s, d
vRngRayleigh	s, d
vRngLognormal	s, d
vRngGamma	s, d
vRngBeta	s, d

See additional details about accurate and fast mode of random number generation in VS Notes.

New method names

The current version of oneMKL has a modified structure of VS RNG method names. (SeeRNG Naming Conventions for details.) The old names are kept for backward compatibility. The tables below set correspondence between the new and legacy method names for VS random number generators.

Method Names for Continuous Distribution Generators

RNG	Legacy Method Name	New Method Name
vRngUniform	VSL_METHOD_SUNIFORM_STD, VSL_METHOD_DUNIFORM_STD, VSL_METHOD_SUNIFORM_STD_ACCURATE, VSL_METHOD_DUNIFORM_STD_ACCURATE	VSL_RNG_METHOD_UNIFORM_STD, VSL_RNG_METHOD_UNIFORM_STD_ACCURATE
vRngGaussian	VSL_METHOD_SGAUSSIAN_BOXMULLER, VSL_METHOD_SGAUSSIAN_BOXMULLER2, VSL_METHOD_SGAUSSIAN_ICDF, VSL_METHOD_DGAUSSIAN_BOXMULLER, VSL_METHOD_DGAUSSIAN_BOXMULLER2, VSL_METHOD_DGAUSSIAN_ICDF	VSL_RNG_METHOD_GAUSSIAN_BOXMULLER, VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2, VSL_RNG_METHOD_GAUSSIAN_ICDF
vRngGaussianMV	VSL_METHOD_SGAUSSIANMV_BOXMULLER, VSL_METHOD_SGAUSSIANMV_BOXMULLER2, VSL_METHOD_SGAUSSIANMV_ICDF, VSL_METHOD_DGAUSSIANMV_BOXMULLER, VSL_METHOD_DGAUSSIANMV_BOXMULLER2, VSL_METHOD_DGAUSSIANMV_ICDF	VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER , VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER 2, VSL_RNG_METHOD_GAUSSIANMV_ICDF
vRngExponential	VSL_METHOD_SEXPONENTIAL_ICDF, VSL_METHOD_DEXPONENTIAL_ICDF, VSL_METHOD_SEXPONENTIAL_ICDF_ACCUR ATE, VSL_METHOD_DEXPONENTIAL_ICDF_ACCUR ATE	VSL_RNG_METHOD_EXPONENTIAL_ICDF, VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACC URATE
vRngLaplace	VSL_METHOD_SLAPLACE_ICDF, VSL_METHOD_DLAPLACEL_ICDF	VSL_RNG_METHOD_LAPLACE_ICDF
vRngWeibull	VSL_METHOD_SWEIBULL_ICDF, VSL_METHOD_DWEIBULL_ICDF, VSL_METHOD_SWEIBULL_ICDF_ACCURATE, VSL_METHOD_DWEIBULL_ICDF_ACCURATE	VSL_RNG_METHOD_WEIBULL_ICDF, VSL_RNG_METHOD_WEIBULL_ICDF_ACCURAT E

RNG	Legacy Method Name	New Method Name
vRngCauchy	VSL_METHOD_SCAUCHY_ICDF, VSL_METHOD_DCAUCHY_ICDF	VSL_RNG_METHOD_CAUCHY_ICDF
vRngRayleigh	VSL_METHOD_SRAYLEIGH_ICDF, VSL_METHOD_DRAYLEIGH_ICDF, VSL_METHOD_SRAYLEIGH_ICDF_ACCURATE, VSL_METHOD_DRAYLEIGH_ICDF_ACCURATE	VSL_RNG_METHOD_RAYLEIGH_ICDF, VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE
vRngLognormal	VSL_METHOD_SLOGNORMAL_BOXMULLER2, VSL_METHOD_DLOGNORMAL_BOXMULLER2, VSL_METHOD_SLOGNORMAL_BOXMULLER2_ACCURATE, VSL_METHOD_DLOGNORMAL_BOXMULLER2_ACCURATE	VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2, VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE
	VSL_METHOD_SLOGNORMAL_ICDF, VSL_METHOD_DLOGNORMAL_ICDF, VSL_METHOD_SLOGNORMAL_ICDF_ACCURATE, VSL_METHOD_DLOGNORMAL_ICDF_ACCURATE	VSL_RNG_METHOD_LOGNORMAL_ICDF, VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE
vRngGumbel	VSL_METHOD_SGUMBEL_ICDF, VSL_METHOD_DGUMBEL_ICDF	VSL_RNG_METHOD_GUMBEL_ICDF
vRngGamma	VSL_METHOD_SGAMMA_GNORM, VSL_METHOD_DGAMMA_GNORM, VSL_METHOD_SGAMMA_GNORM_ACCURATE, VSL_METHOD_DGAMMA_GNORM_ACCURATE	VSL_RNG_METHOD_GAMMA_GNORM, VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE
vRngBeta	VSL_METHOD_SBETA_CJA, VSL_METHOD_DBETA_CJA, VSL_METHOD_SBETA_CJA_ACCURATE, VSL_METHOD_DBETA_CJA_ACCURATE	VSL_RNG_METHOD_BETA_CJA, VSL_RNG_METHOD_BETA_CJA_ACCURATE

Method Names for Discrete Distribution Generators

RNG	Legacy Method Name	New Method Name
vRngUniform	VSL_METHOD_IUNIFORM_STD	VSL_RNG_METHOD_UNIFORM_STD
vRngUniformBits	VSL_METHOD_IUNIFORMBITS_STD	VSL_RNG_METHOD_UNIFORMBITS_STD
vRngBernoulli	VSL_METHOD_IBERNOULLI_ICDF	VSL_RNG_METHOD_BERNOULLI_ICDF
vRngGeometric	VSL_METHOD_IGEOMETRIC_ICDF	VSL_RNG_METHOD_GEOMETRIC_ICDF
vRngBinomial	VSL_METHOD_IBINOMIAL_BTPE	VSL_RNG_METHOD_BINOMIAL_BTPE
vRngHypergeometric	VSL_METHOD_IHYPERGEOMETRIC_H2PE	VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE
vRngPoisson	VSL_METHOD_IPOISSON_PTPE, VSL_METHOD_IPOISSON_POISNORM	VSL_RNG_METHOD_POISSON_PTPE, VSL_RNG_METHOD_POISSON_POISNORM
vRngPoissonV	VSL_METHOD_IPOISSONV_POISNORM	VSL_RNG_METHOD_POISSONV_POISNORM
vRngNegBinomial	VSL_METHOD_INEGBINOMIAL_NBAR	VSL_RNG_METHOD_NEGBINOMIAL_NBAR

Continuous Distributions

This section describes routines for generating random numbers with continuous distribution.

`vRngUniform` *Continuous Distribution Generators*

Generates random numbers with uniform distribution.

Syntax

```
status = vsrnguniform( method, stream, n, r, a, b )
status = vdrnguniform( method, stream, n, r, a, b )
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Description

The `vRngUniform` function generates random numbers uniformly distributed over the interval `[a, b)`, where `a, b` are the left and right bounds of the interval, respectively, and `a, b∈R ; a < b`.

The probability density function is given by:

$$f_{a,b}(x) = \begin{cases} \dfrac{1}{b-a}, & x \in [a,b) \\ 0, & x \notin [a,b) \end{cases}, \quad -\infty < x < +\infty$$

The cumulative distribution function is as follows:

$$f_{a,b}(x) = \begin{cases} 0, & x < a \\ \dfrac{x-a}{b-a}, & a \le x < b \\ 1, & x \ge b \end{cases}, \quad -\infty < x < +\infty .$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

Name	Type	Description

method	INTEGER, INTENT(IN)	Generation method; the specific values are as follows:
		`VSL_RNG_METHOD_UNIFORM_STD`
		`VSL_RNG_METHOD_UNIFORM_STD_ACCURATE`
		Standard method.

Name	Type	Description
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated.
a	DOUBLE PRECISION for vdrnguniform REAL (KIND=4), INTENT(IN) for vsrnguniform REAL (KIND=8), INTENT(IN) for vdrnguniform	Left bound a.
b	DOUBLE PRECISION for vdrnguniform REAL (KIND=4), INTENT(IN) for vsrnguniform REAL (KIND=8), INTENT(IN) for vdrnguniform	Right bound b.

Output Parameters

Name	Type	Description
r	DOUBLE PRECISION for vdrnguniform REAL (KIND=4), INTENT(OUT) for vsrnguniform REAL (KIND=8), INTENT(OUT) for vdrnguniform	Vector of n random numbers uniformly distributed over the interval [*a*,*b*)

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax .
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGaussian
Generates normally distributed random numbers.

Syntax

status = vsrnggaussian(*method*, *stream*, *n*, *r*, *a*, *sigma*)

status = vdrnggaussian(*method*, *stream*, *n*, *r*, *a*, *sigma*)

Include Files

- mkl.fi, mkl_vsl.f90

Description

The vRngGaussian function generates random numbers with normal (Gaussian) distribution with mean value a and standard deviation σ, where

$a, \sigma \in R ; \sigma > 0.$

The probability density function is given by:

$$f_{a,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-a)^2}{2\sigma^2}\right), \quad -\infty < x < +\infty .$$

The cumulative distribution function is as follows:

$$F_{a,\sigma}(x) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-a)^2}{2\sigma^2}\right) dy, \quad -\infty < x < +\infty .$$

The cumulative distribution function $F_{a,\sigma}(x)$ can be expressed in terms of standard normal distribution $\Phi(x)$ as

$F_{a,\sigma}(x) = \Phi((x - a)/\sigma)$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Input Parameters

Name	Type	Description

method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows:
		`VSL_RNG_METHOD_GAUSSIAN_BOXMULLER`
		`VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2`
		`VSL_RNG_METHOD_GAUSSIAN_ICDF`
		See brief description of the methods `BOXMULLER`, `BOXMULLER2`, and `ICDF` in Table "Values of <*method*> in *method* parameter"

stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure
n	INTEGER, INTENT(IN)	Number of random values to be generated.
a	DOUBLE PRECISION for vdrnggaussian REAL (KIND=4), INTENT(IN) for vsrnggaussian REAL (KIND=8), INTENT(IN) for vdrnggaussian	Mean value a.
sigma	DOUBLE PRECISION for vdrnggaussian REAL (KIND=4), INTENT(IN) for vsrnggaussian REAL (KIND=8), INTENT(IN) for vdrnggaussian	Standard deviation σ.

Output Parameters

Name	Type	Description
r	DOUBLE PRECISION for vdrnggaussian REAL (KIND=4), INTENT(OUT) for vsrnggaussian	Vector of `n` normally distributed random numbers.

Name	Type	Description
	REAL (KIND=8), INTENT(OUT) for vdrnggaussian	

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGaussianMV
Generates random numbers from multivariate normal distribution.

Syntax

status = vsrnggaussianmv(*method*, *stream*, *n*, *r*, *dimen*, *mstorage*, *a*, *t*)

status = vdrnggaussianmv(*method*, *stream*, *n*, *r*, *dimen*, *mstorage*, *a*, *t*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2 VSL_RNG_METHOD_GAUSSIANMV_ICDF See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table "Values of <*method*> in *method* parameter"

Name	Type	Description
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of *d*-dimensional vectors to be generated
dimen	INTEGER, INTENT(IN)	Dimension d ($d \geq 1$) of output random vectors
mstorage	INTEGER, INTENT(IN)	Matrix storage scheme for upper triangular matrix T^T. The routine supports three matrix storage schemes: • VSL_MATRIX_STORAGE_FULL— all d x d elements of the matrix T^T are passed, however, only the upper triangle part is actually used in the routine. • VSL_MATRIX_STORAGE_PACKED— upper triangle elements of T^T are packed by rows into a one-dimensional array. • VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of T^T are passed.
a	DOUBLE PRECISION for vdrnggaussianmv REAL(KIND=4), INTENT(IN) for vsrnggaussianmv REAL(KIND=8), INTENT(IN) for vdrnggaussianmv	Mean vector *a* of dimension *d*
t	DOUBLE PRECISION for vdrnggaussianmv REAL(KIND=4), INTENT(IN) for vsrnggaussianmv REAL(KIND=8), INTENT(IN) for vdrnggaussianmv	Elements of the upper triangular matrix passed according to the matrix T^T storage scheme *mstorage*.

Output Parameters

Name	Type	Description
r	DOUBLE PRECISION for vdrnggaussianmv REAL(KIND=4), INTENT(OUT) for vsrnggaussianmv REAL(KIND=8), INTENT(OUT) for vdrnggaussianmv	Array of *n* random vectors of dimension *dimen*

Description

The vRngGaussianMV function generates random numbers with *d*-variate normal (Gaussian) distribution with mean value a and variance-covariance matrix *C*, where $a \in R^d$; C is a $d \times d$ symmetric positive-definite matrix.

The probability density function is given by:

$$f_{a,C}(x) = \frac{1}{\sqrt{\det(2\pi C)}} \exp\left(-1/2(x-a)^T C^{-1}(x-a)\right),$$

where $x \in R^d$.

Matrix *C* can be represented as $C = TT^T$, where *T* is a lower triangular matrix - Cholesky factor of *C*.

Instead of variance-covariance matrix *C* the generation routines require Cholesky factor of *C* in input. To compute Cholesky factor of matrix *C*, the user may call Intel® oneAPI Math Kernel Library (oneMKL) LAPACK routines for matrix factorization: `?potrf` or `?pptrf` for `v?RngGaussianMV`/`v?rnggaussianmv` routines (`?` means either `s` or `d` for single and double precision respectively). See Application Notes for more details.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Application Notes

Since matrices are stored in Fortran by columns, while in C they are stored by rows, the usage of Intel® oneAPI Math Kernel Library (oneMKL) factorization routines (assuming Fortran matrices storage) in combination with multivariate normal RNG (assuming C matrix storage) is slightly different in C and Fortran. The following tables help in using these routines in C and Fortran. For further information please refer to the appropriate VS example file.

Using Cholesky Factorization Routines in Fortran

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Routine	UPLO Parameter in Factorization Routine	Result of Factorization as Input Argument for RNG
`VSL_MATRIX_STORAGE_FULL`	*C* in Fortran two-dimensional array	`spotrf` for `vsrnggaussianmv` `dpotrf` for `vdrnggaussianmv`	'U'	Upper triangle of T^T. Lower triangle is not used.
`VSL_MATRIX_STORAGE_PACKED`	Lower triangle of *C* packed by columns into one-dimensional array	`spptrf` for `vsrnggaussianmv` `dpptrf` for `vdrnggaussianmv`	'L'	Upper triangle of T^T packed by rows into a one-dimensional array.

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*stream* is a `NULL` pointer.
`VSL_RNG_ERROR_BAD_STREAM`	*stream* is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngExponential
Generates exponentially distributed random numbers.

Syntax

status = vsrngexponential(*method*, *stream*, *n*, *r*, *a*, *beta*)

status = vdrngexponential(*method*, *stream*, *n*, *r*, *a*, *beta*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_EXPONENTIAL_ICDF VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE Inverse cumulative distribution function method
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
a	DOUBLE PRECISION for vdrngexponential REAL(KIND=4), INTENT(IN) for vsrngexponential REAL(KIND=8), INTENT(IN) for vdrngexponential	Displacement *a*
beta	DOUBLE PRECISION for vdrngexponential	Scalefactor β.

Name	Type	Description
	`REAL(KIND=4), INTENT(IN)` for `vsrngexponential` `REAL(KIND=8), INTENT(IN)` for `vdrngexponential`	

Output Parameters

Name	Type	Description

r	`DOUBLE PRECISION` for `vdrngexponential`
	`REAL(KIND=4), INTENT(OUT)` for `vsrngexponential`
	`REAL(KIND=8), INTENT(OUT)` for `vdrngexponential`
		Vector of *n* exponentially distributed random numbers

Description

The `vRngExponential` function generates random numbers with exponential distribution that has displacement `a` and scalefactor β, where $a,\ \beta \in R\ ;\ \beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} \exp\left((-(x-a)) / \beta \right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left((-(x-a)) / \beta \right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*`stream`* is a `NULL` pointer.
`VSL_RNG_ERROR_BAD_STREAM`	*stream* is not a valid random stream.
`VSL_RNG_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.

VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngLaplace
Generates random numbers with Laplace distribution.

Syntax

status = vsrnglaplace(*method*, *stream*, *n*, *r*, *a*, *beta*)

status = vdrnglaplace(*method*, *stream*, *n*, *r*, *a*, *beta*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_LAPLACE_ICDF Inverse cumulative distribution function method
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
a	DOUBLE PRECISION for vdrnglaplace REAL(KIND=4), INTENT(IN) for vsrnglaplace REAL(KIND=8), INTENT(IN) for vdrnglaplace	Mean value *a*
beta	DOUBLE PRECISION for vdrnglaplace REAL(KIND=4), INTENT(IN) for vsrnglaplace REAL(KIND=8), INTENT(IN) for vdrnglaplace	Scalefactor β.

Output Parameters

Name	Type	Description
r	DOUBLE PRECISION for vdrnglaplace REAL(KIND=4), INTENT(OUT) for vsrnglaplace REAL(KIND=8), INTENT(OUT) for vdrnglaplace	Vector of *n* Laplace distributed random numbers

Description

The vRngLaplace function generates random numbers with Laplace distribution with mean value (or average) *a* and scalefactor β, where $a,\ \beta \in R\ ;\ \beta > 0$. The scalefactor value determines the standard deviation as

$$\sigma = \beta\sqrt{2}$$

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\sqrt{2}\beta} \exp\left(-\frac{|x-a|}{\beta}\right), -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{2}\exp\left(-\frac{|x-a|}{\beta}\right), & x \geq a \\ 1 - \frac{1}{2}\exp\left(-\frac{|x-a|}{\beta}\right), & x < a \end{cases}, -\infty < x < +\infty.$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.

VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngWeibull
Generates Weibull distributed random numbers.

Syntax

status = vsrngweibull(*method*, *stream*, *n*, *r*, *alpha*, *a*, *beta*)

status = vdrngweibull(*method*, *stream*, *n*, *r*, *alpha*, *a*, *beta*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_WEIBULL_ICDF VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE Inverse cumulative distribution function method
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
alpha	DOUBLE PRECISION for vdrngweibull REAL(KIND=4), INTENT(IN) for vsrngweibull REAL(KIND=8), INTENT(IN) for vdrngweibull	Shape α.
a	DOUBLE PRECISION for vdrngweibull REAL(KIND=4), INTENT(IN) for vsrngweibull REAL(KIND=8), INTENT(IN) for vdrngweibull	Displacement *a*

Name	Type	Description

beta	`DOUBLE PRECISION` for `vdrngweibull`
	`REAL(KIND=4), INTENT(IN)` for `vsrngweibull`
	`REAL(KIND=8), INTENT(IN)` for `vdrngweibull`
		Scalefactor β.

Output Parameters

Name	Type	Description

r	`DOUBLE PRECISION` for `vdrngweibull`
	`REAL(KIND=4), INTENT(OUT)` for `vsrngweibull`
	`REAL(KIND=8), INTENT(OUT)` for `vdrngweibull`
		Vector of *n* Weibull distributed random numbers

Description

The `vRngWeibull` function generates Weibull distributed random numbers with displacement *a*, scalefactor β, and shape α, where α, β, *a*∈*R* ; α > 0, β > 0.

The probability density function is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{\alpha}{\beta^{\alpha}}(x-a)^{\alpha-1}\exp\left(-\left(\frac{x-a}{\beta}\right)^{\alpha}\right), & x \geq a \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} 1-\exp\left(-\left(\frac{x-a}{\beta}\right)^{\alpha}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngCauchy
Generates Cauchy distributed random values.

Syntax

status = vsrngcauchy(*method*, *stream*, *n*, *r*, *a*, *beta*)

status = vdrngcauchy(*method*, *stream*, *n*, *r*, *a*, *beta*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_CAUCHY_ICDF Inverse cumulative distribution function method
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
a	DOUBLE PRECISION for vdrngcauchy REAL(KIND=4), INTENT(IN) for vsrngcauchy REAL(KIND=8), INTENT(IN) for vdrngcauchy	Displacement *a*.
beta	DOUBLE PRECISION for vdrngcauchy	Scalefactor β.

Name	Type	Description
	`REAL(KIND=4), INTENT(IN)` for `vsrngcauchy` `REAL(KIND=8), INTENT(IN)` for `vdrngcauchy`	

Output Parameters

Name	Type	Description

r	`DOUBLE PRECISION` for `vdrngcauchy`
	`REAL(KIND=4), INTENT(OUT)` for `vsrngcauchy`
	`REAL(KIND=8), INTENT(OUT)` for `vdrngcauchy`
		Vector of *n* Cauchy distributed random numbers

Description

The function generates Cauchy distributed random numbers with displacement *a* and scalefactor β, where $a, \beta \in R ; \beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta\left(1 + \left(\frac{x-a}{\beta}\right)^2\right)}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi}\arctan\left(\frac{x-a}{\beta}\right), -\infty < x < +\infty.$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*stream* is a `NULL` pointer.
`VSL_RNG_ERROR_BAD_STREAM`	*stream* is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngRayleigh
Generates Rayleigh distributed random values.

Syntax

status = vsrngrayleigh(*method*, *stream*, *n*, *r*, *a*, *beta*)

status = vdrngrayleigh(*method*, *stream*, *n*, *r*, *a*, *beta*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_RAYLEIGH_ICDF VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE Inverse cumulative distribution function method
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
a	DOUBLE PRECISION for vdrngrayleigh REAL(KIND=4), INTENT(IN) for vsrngrayleigh REAL(KIND=8), INTENT(IN) for vdrngrayleigh	Displacement *a*
beta	DOUBLE PRECISION for vdrngrayleigh REAL(KIND=4), INTENT(IN) for vsrngrayleigh	Scalefactor β.

Name	Type	Description
	`REAL(KIND=8), INTENT(IN)` for `vdrngrayleigh`	

Output Parameters

Name	Type	Description

r	`DOUBLE PRECISION` for `vdrngrayleigh`
	`REAL(KIND=4), INTENT(OUT)` for `vsrngrayleigh`
	`REAL(KIND=8), INTENT(OUT)` for `vdrngrayleigh`
		Vector of *n* Rayleigh distributed random numbers

Description

The `vRngRayleigh` function generates Rayleigh distributed random numbers with displacement *a* and scalefactor β, where $a, \beta \in R ; \beta > 0$.

The Rayleigh distribution is a special case of the `Weibull` distribution, where the shape parameter $\alpha = 2$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x-a)}{\beta^2} \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*`stream`* is a `NULL` pointer.
`VSL_RNG_ERROR_BAD_STREAM`	*stream* is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngLognormal
Generates lognormally distributed random numbers.

Syntax

status = vsrnglognormal(*method*, *stream*, *n*, *r*, *a*, *sigma*, *b*, *beta*)

status = vdrnglognormal(*method*, *stream*, *n*, *r*, *a*, *sigma*, *b*, *beta*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2 VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE Box Muller 2 based method VSL_RNG_METHOD_LOGNORMAL_ICDF VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE Inverse cumulative distribution function based method
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
a	DOUBLE PRECISION for vdrnglognormal REAL(KIND=4), INTENT(IN) for vsrnglognormal REAL(KIND=8), INTENT(IN) for vdrnglognormal	Average *a* of the subject normal distribution

Name	Type	Description

sigma	`DOUBLE PRECISION` for `vdrnglognormal`
	`REAL(KIND=4), INTENT(IN)` for `vsrnglognormal`
	`REAL(KIND=8), INTENT(IN)` for `vdrnglognormal`
		Standard deviation σ of the subject normal distribution

b	`DOUBLE PRECISION` for `vdrnglognormal`
	`REAL(KIND=4), INTENT(IN)` for `vsrnglognormal`
	`REAL(KIND=8), INTENT(IN)` for `vdrnglognormal`
		Displacement *b*

beta	`DOUBLE PRECISION` for `vdrnglognormal`
	`REAL(KIND=4), INTENT(IN)` for `vsrnglognormal`
	`REAL(KIND=8), INTENT(IN)` for `vdrnglognormal`
		Scalefactor β.

Output Parameters

Name	Type	Description

r	`DOUBLE PRECISION` for `vdrnglognormal`
	`REAL(KIND=4), INTENT(OUT)` for `vsrnglognormal`
	`REAL(KIND=8), INTENT(OUT)` for `vdrnglognormal`
		Vector of *n* lognormally distributed random numbers

Description

The `vRngLognormal` function generates lognormally distributed random numbers with average of distribution *a* and standard deviation σ of subject normal distribution, displacement *b*, and scalefactor β, where a, σ, b, $\beta \in R$; $\sigma > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \dfrac{1}{\sigma(x-b)\sqrt{2\pi}} \exp\left(-\dfrac{\left[\ln((x-b)/\beta) - a\right]^2}{2\sigma^2}\right), & x > b \\ 0, & x \le b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi(\ (\ln(\ (x - b) / \beta) - a)/\sigma)\ , & x > b \\ 0, & x \le b \end{cases}$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGumbel
Generates Gumbel distributed random values.

Syntax

status = vsrnggumbel(*method*, *stream*, *n*, *r*, *a*, *beta*)

status = vdrnggumbel(*method*, *stream*, *n*, *r*, *a*, *beta*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_GUMBEL_ICDF Inverse cumulative distribution function method

Name	Type	Description
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure
n	INTEGER, INTENT(IN)	Number of random values to be generated
a	DOUBLE PRECISION for vdrnggumbel REAL(KIND=4), INTENT(IN) for vsrnggumbel REAL(KIND=8), INTENT(IN) for vdrnggumbel	Displacement a.
beta	DOUBLE PRECISION for vdrnggumbel REAL(KIND=4), INTENT(IN) for vsrnggumbel REAL(KIND=8), INTENT(IN) for vdrnggumbel	Scalefactor β.

Output Parameters

Name	Type	Description
r	DOUBLE PRECISION for vdrnggumbel REAL(KIND=4), INTENT(OUT) for vsrnggumbel REAL(KIND=8), INTENT(OUT) for vdrnggumbel	Vector of *n* random numbers with Gumbel distribution

Description

The vRngGumbel function generates Gumbel distributed random numbers with displacement a and scalefactor β, where a, β∈*R* ; β > 0.

The probability density function is given by:

$$f_{a,\beta}(x) = \left\{\frac{1}{\beta}\exp\left(\frac{x-a}{\beta}\right)\exp(-\exp((x-a)/\beta)), -\infty < x < +\infty\right.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = 1 - \exp(-\exp((x-a)/\beta)), -\infty < x < +\infty$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Product and Performance Information
Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGamma
Generates gamma distributed random values.

Syntax

status = vsrnggamma(*method*, *stream*, *n*, *r*, *alpha*, *a*, *beta*)

status = vdrnggamma(*method*, *stream*, *n*, *r*, *alpha*, *a*, *beta*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_GAMMA_GNORM VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE Acceptance/rejection method using random numbers with Gaussian distribution. See brief description of the method GNORM in Table "Values of <*method*> in *method* parameter"
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure
n	INTEGER, INTENT(IN)	Number of random values to be generated

Name	Type	Description

alpha	`DOUBLE PRECISION` for `vdrnggamma`
	`REAL(KIND=4), INTENT(IN)` for `vsrnggamma`
	`REAL(KIND=8), INTENT(IN)` for `vdrnggamma`
		Shape α.

a	`DOUBLE PRECISION` for `vdrnggamma`
	`REAL(KIND=4), INTENT(IN)` for `vsrnggamma`
	`REAL(KIND=8), INTENT(IN)` for `vdrnggamma`
		Displacement *a*.

beta	`DOUBLE PRECISION` for `vdrnggamma`
	`REAL(KIND=4), INTENT(IN)` for `vsrnggamma`
	`REAL(KIND=8), INTENT(IN)` for `vdrnggamma`
		Scalefactor β.

Output Parameters

Name	Type	Description

r	`DOUBLE PRECISION` for `vdrnggamma`
	`REAL(KIND=4), INTENT(OUT)` for `vsrnggamma`
	`REAL(KIND=8), INTENT(OUT)` for `vdrnggamma`
		Vector of *n* random numbers with gamma distribution

Description

The `vRngGamma` function generates random numbers with gamma distribution that has shape parameter α, displacement *a*, and scale parameter β, where α, β, and $a \in R$; $\alpha > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{\alpha,a,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^{\alpha}}(x-a)^{\alpha-1}e^{-(x-a)/\beta}, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

where Γ(α) is the complete gamma function.

The cumulative distribution function is as follows:

$$F_{\alpha,a,\beta}(x) = \begin{cases} \int_a^x \frac{1}{\Gamma(\alpha)\beta^\alpha}(y-a)^{\alpha-1}e^{-(y-a)/\beta}dy, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngBeta
Generates beta distributed random values.

Syntax

status = vsrngbeta(*method*, *stream*, *n*, *r*, *p*, *q*, *a*, *beta*)

status = vdrngbeta(*method*, *stream*, *n*, *r*, *p*, *q*, *a*, *beta*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_BETA_CJA VSL_RNG_METHOD_BETA_CJA_ACCURATE See brief description of the method CJA in Table "Values of <method> in *method* parameter"
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure
n	INTEGER, INTENT(IN)	Number of random values to be generated
p	DOUBLE PRECISION for vdrngbeta REAL(KIND=4), INTENT(IN) for vsrngbeta REAL(KIND=8), INTENT(IN) for vdrngbeta	Shape *p*
q	DOUBLE PRECISION for vdrngbeta REAL(KIND=4), INTENT(IN) for vsrngbeta REAL(KIND=8), INTENT(IN) for vdrngbeta	Shape *q*
a	DOUBLE PRECISION for vdrngbeta REAL(KIND=4), INTENT(IN) for vsrngbeta REAL(KIND=8), INTENT(IN) for vdrngbeta	Displacement*a*.
beta	DOUBLE PRECISION for vdrngbeta REAL(KIND=4), INTENT(IN) for vsrngbeta REAL(KIND=8), INTENT(IN) for vdrngbeta	Scalefactor β.

Output Parameters

Name	Type	Description
r	DOUBLE PRECISION for vdrngbeta	Vector of *n* random numbers with beta distribution

Name	Type	Description
	`REAL(KIND=4), INTENT(OUT)` for `vsrngbeta` `REAL(KIND=8), INTENT(OUT)` for `vdrngbeta`	

Description

The `vRngBeta` function generates random numbers with beta distribution that has shape parameters `p` and *q*, displacement *a*, and scale parameter β, where *p*, *q*, *a*, and β∈R ; $p > 0$, $q > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{p,q,a,\beta}(x) = \begin{cases} \frac{1}{B(p,q)\beta^{p+q-1}}(x-a)^{p-1}(\beta+a-x)^{q-1}, & a \le x < a+\beta \\ 0, & x < a, x \ge a+\beta \end{cases}, -\infty < x <$$

where $B(p, q)$ is the complete beta function.

The cumulative distribution function is as follows:

$$F_{p,q,a,\beta}(x) = \begin{cases} 0, & x < a \\ \int_a^x \frac{1}{B(p,q)\beta^{p+q-1}}(y-a)^{p-1}(\beta+a-y)^{q-1}dy, & a \le x < a+\beta \\ 1, & x \ge a+\beta \end{cases}, -\infty < x$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*`stream`* is a `NULL` pointer.
`VSL_RNG_ERROR_BAD_STREAM`	*stream* is not a valid random stream.
`VSL_RNG_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.
`VSL_RNG_ERROR_NO_NUMBERS`	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
`VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED`	Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngChiSquare
Generates chi-square distributed random values.

Syntax

status = vsrngchisquare(*method*, *stream*, *n*, *r*, *v*)

status = vdrngchisquare(*method*, *stream*, *n*, *r*, *v*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific value is: VSL_RNG_METHOD_CHISQUARE_CHI2GAMMA For a description of VSL_RNG_METHOD_CHISQUARE_CHI2GAMMA, see Random Number Generators Naming Conventions.
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure
n	INTEGER, INTENT(IN)	Number of random values to be generated
v	INTEGER, INTENT(IN)	Degrees of freedom

Output Parameters

Name	Type	Description
r	REAL(KIND=4), INTENT(OUT) for vsrngchisquare REAL(KIND=8), INTENT(OUT) for vdrngchisquare	Vector of *n* random numbers with chi-square distribution

Description

The vRngChiSquare function generates random numbers with chi-square distribution and *v* degrees of freedom, $v \in N$, $v > 0$.

The probability density function is:

$$f_v(x) = \begin{cases} \frac{x^{\frac{v-2}{2}} e^{-\frac{x}{2}}}{2^{v/2}\Gamma\left(\frac{v}{2}\right)}, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

The cumulative distribution function is:

$$F_v(x) = \begin{cases} \int_0^x \frac{y^{\frac{v-2}{2}} e^{-\frac{y}{2}}}{2^{v/2}\Gamma\left(\frac{v}{2}\right)} dy, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*`stream`* is a `NULL` pointer.
`VSL_RNG_ERROR_BAD_STREAM`	*stream* is not a valid random stream.
`VSL_RNG_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.
`VSL_RNG_ERROR_NO_NUMBERS`	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
`VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED`	Period of the generator has been exceeded.
`VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED`	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
`VSL_RNG_ERROR_ARS5_NOT_SUPPORTED`	ARS-5 random number generator is not supported on the CPU running the application.

Discrete Distributions

This section describes routines for generating random numbers with discrete distribution.

`vRngUniform` *Discrete Distribution Generators*

Generates random numbers uniformly distributed over the interval `[a, b)`.

Syntax

```
status = virnguniform( method, stream, n, r, a, b )
```

Include Files

- `mkl.fi, mkl_vsl.f90`

Input Parameters

Name	Type	Description
`method`	`INTEGER, INTENT(IN)`	Generation method; the specific value is as follows: `VSL_RNG_METHOD_UNIFORM_STD`

Name	Type	Description
		Standard method. Currently there is only one method for this distribution generator.
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
a	INTEGER(KIND=4), INTENT(IN)	Left interval bound a
b	INTEGER(KIND=4), INTENT(IN)	Right interval bound b

Output Parameters

Name	Type	Description
r	INTEGER(KIND=4), INTENT(OUT)	Vector of *n* random numbers uniformly distributed over the interval [*a*,*b*)

Description

The vRngUniform function generates random numbers uniformly distributed over the interval [*a*, *b*), where *a*, *b* are the left and right bounds of the interval respectively, and *a*, *b*∈*Z*; *a* < *b*.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b - a}, k \in \{a, a + 1, \ldots, b - 1\}.$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \dfrac{\lfloor x - a + 1 \rfloor}{b - a}, & a \le x < b, x \in R. \\ 1, & x \ge b \end{cases}$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits
Generates bits of underlying BRNG integer recurrence.

Syntax

status = virnguniformbits(*method*, *stream*, *n*, *r*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method; the specific value is VSL_RNG_METHOD_UNIFORMBITS_STD
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated

Output Parameters

Name	Type	Description
r	INTEGER(KIND=4), INTENT(OUT)	Vector of *n* random integer numbers. If the *stream* was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of *r* respectively. The number of bytes occupied by each integer is contained in the field *wordsize* of the structure VSL_BRNG_PROPERTIES. The total number of bits that are actually used to store the value are contained in the field *nbits* of the same structure. See Advanced Service Routines for a more detailed discussion of VSLBRngProperties.

Description

The `vRngUniformBits` function generates integer random values with uniform bit distribution. The generators of uniformly distributed numbers can be represented as recurrence relations over integer values in modular arithmetic. Apparently, each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated. For example, a well known drawback of linear congruential generators is that lower bits are less random than higher bits (for example, see [Knuth81]). For this reason, care should be taken when using this function. Typically, in a 32-bit *LCG* only 24 higher bits of an integer value can be considered random. See VS Notes for details.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*`stream`* is a `NULL` pointer.
`VSL_RNG_ERROR_BAD_STREAM`	*`stream`* is not a valid random stream.
`VSL_RNG_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.
`VSL_RNG_ERROR_NO_NUMBERS`	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
`VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED`	Period of the generator has been exceeded.
`VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED`	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
`VSL_RNG_ERROR_ARS5_NOT_SUPPORTED`	ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits32
Generates uniformly distributed bits in 32-bit chunks.

Syntax

```
status = virnguniformbits32( method, stream, n, r )
```

Include Files

- `mkl.fi, mkl_vsl.f90`

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method; the specific value is VSL_RNG_METHOD_UNIFORMBITS32_STD
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated

Output Parameters

Name	Type	Description
r	INTEGER (KIND=4), INTENT(OUT)	Vector of *n* 32-bit random integer numbers with uniform bit distribution.

Description

The vRngUniformBits32 function generates uniformly distributed bits in 32-bit chunks. Unlike vRngUniformBits, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, vRngUniformBits32 is designed to ensure each bit in the 32-bit chunk is uniformly distributed. See VS Notes for details.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits64
Generates uniformly distributed bits in 64-bit chunks.

Syntax

```
status = virnguniformbits64( method, stream, n, r )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method; the specific value is VSL_RNG_METHOD_UNIFORMBITS64_STD
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated

Output Parameters

Name	Type	Description
r	INTEGER (KIND=8), INTENT(OUT)	Vector of *n* 64-bit random integer numbers with uniform bit distribution.

Description

The vRngUniformBits64 function generates uniformly distributed bits in 64-bit chunks. Unlike vRngUniformBits, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, vRngUniformBits64 is designed to ensure each bit in the 64-bit chunk is uniformly distributed. See VS Notes for details.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngBernoulli
Generates Bernoulli distributed random values.

Syntax

```
status = virngbernoulli( method, stream, n, r, p )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific value is as follows: VSL_RNG_METHOD_BERNOULLI_ICDF Inverse cumulative distribution function method.
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
p	REAL(KIND=8), INTENT(IN)	Success probability *p* of a trial

Output Parameters

Name	Type	Description
r	INTEGER(KIND=4), INTENT(OUT)	Vector of *n* Bernoulli distributed random values

Description

The vRngBernoulli function generates Bernoulli distributed random numbers with probability *p* of a single trial success, where

$p \in R;\ 0 \le p \le 1.$

A variate is called Bernoulli distributed, if after a trial it is equal to 1 with probability of success *p*, and to 0 with probability 1 - *p*.

The probability distribution is given by:

$P(X = 1) = p$

$P(X = 0) = 1 - p$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - p, & 0 \le x < 1, x \in R. \\ 1, & x \ge 1 \end{cases}$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGeometric
Generates geometrically distributed random values.

Syntax

```
status = virnggeometric( method, stream, n, r, p )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific value is as follows: VSL_RNG_METHOD_GEOMETRIC_ICDF Inverse cumulative distribution function method.
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
p	REAL(KIND=8), INTENT(IN)	Success probability *p* of a trial

Output Parameters

Name	Type	Description
r	INTEGER(KIND=4), INTENT(OUT)	Vector of *n* geometrically distributed random values

Description

The vRngGeometric function generates geometrically distributed random numbers with probability *p* of a single trial success, where *p*∈*R*; 0 < *p* < 1.

A geometrically distributed variate represents the number of independent Bernoulli trials preceding the first success. The probability of a single Bernoulli trial success is *p*.

The probability distribution is given by:

$P(X = k) = p \cdot (1 - p)^k$, $k \in \{0,1,2, \ldots\}$.

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x+1 \rfloor}, & 0 \geq x \end{cases} \qquad x \in R.$$

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngBinomial
Generates binomially distributed random numbers.

Syntax

`status = virngbinomial( method, stream, n, r, ntrial, p )`

Include Files

- `mkl.fi, mkl_vsl.f90`

Input Parameters

Name	Type	Description
method	`INTEGER, INTENT(IN)`	Generation method. The specific value is as follows: `VSL_RNG_METHOD_BINOMIAL_BTPE` See brief description of the `BTPE` method in Table "Values of <*method*> in *method* parameter".
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`	Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`	Number of random values to be generated
ntrial	`INTEGER(KIND=4), INTENT(IN)`	Number of independent trials `m`
p	`REAL(KIND=8), INTENT(IN)`	Success probability *p* of a single trial

Output Parameters

Name	Type	Description
r	`INTEGER(KIND=4), INTENT(OUT)`	Vector of *n* binomially distributed random values

Description

The `vRngBinomial` function generates binomially distributed random numbers with number of independent Bernoulli trials *m*, and with probability *p* of a single trial success, where $p \in R;\ 0 \le p \le 1,\ m \in N$.

A binomially distributed variate represents the number of successes in *m* independent Bernoulli trials with probability of a single trial success *p*.

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1 - p)^{m-k}, k \in \{0, 1, \ldots, m\}.$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, & x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1-p)^{m-k}, & 0 \le x < m, x \in R \\ 1, & x \ge m \end{cases}$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngHypergeometric
Generates hypergeometrically distributed random values.

Syntax

status = virnghypergeometric(*method*, *stream*, *n*, *r*, *l*, *s*, *m*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific value is as follows: `VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE` See brief description of the `H2PE` method in Table "Values of <*method*> in *method* parameter"
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
l	INTEGER(KIND=4), INTENT(IN)	Lot size `l`
s	INTEGER(KIND=4), INTENT(IN)	Size of sampling without replacement `s`
m	INTEGER(KIND=4), INTENT(IN)	Number of marked elements `m`

Output Parameters

Name	Type	Description
r	INTEGER(KIND=4), INTENT(OUT)	Vector of *n* hypergeometrically distributed random values

Description

The `vRngHypergeometric` function generates hypergeometrically distributed random values with lot size *l*, size of sampling *s*, and number of marked elements in the lot *m*, where $l, m, s \in N \cup \{0\}; l \geq \max(s, m)$.

Consider a lot of *l* elements comprising *m* "marked" and *l*-*m* "unmarked" elements. A trial sampling without replacement of exactly *s* elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of *s* elements contains exactly *k* marked elements.

The probability distribution is given by:)

$$P(X = k) = \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}$$

, $k \in \{\max(0, s + m - l), \ldots, \min(s, m)\}$

The cumulative distribution function is as follows:

$$F_{l,s,m}(x) = \begin{cases} 0, & x < \max(0, s+m-l) \\ \sum_{k=\max(0,s+m-l)}^{\lfloor x \rfloor} \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, & \max(0, s+m-l) \le x \le \min(s, m) \\ 1, & x > \min(s, m) \end{cases}$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngPoisson
Generates Poisson distributed random values.

Syntax

status = virngpoisson(*method*, *stream*, *n*, *r*, *lambda*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_POISSON_PTPE VSL_RNG_METHOD_POISSON_POISNORM

Name	Type	Description
		See brief description of the PTPE and POISNORM methods in Table "Values of <method> in method parameter".
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
lambda	REAL(KIND=8), INTENT(IN)	Distribution parameter λ.

Output Parameters

Name	Type	Description
r	INTEGER(KIND=4), INTENT(OUT)	Vector of *n* Poisson distributed random values

Description

The vRng"Poisson function generates Poisson distributed random numbers with distribution parameter λ, where λ∈*R*; λ > 0.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!},$$

k∈ {0, 1, 2, ...}.

The cumulative distribution function is as follows:

$$F_\lambda(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngPoissonV
Generates Poisson distributed random values with varying mean.

Syntax

status = virngpoissonv(*method*, *stream*, *n*, *r*, *lambda*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific value is as follows: VSL_RNG_METHOD_POISSONV_POISNORM See brief description of the POISNORM method in Table "Values of <*method*> in *method* parameter"
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
lambda	REAL(KIND=8), INTENT(IN)	Array of *n* distribution parameters λ_i.

Output Parameters

Name	Type	Description
r	INTEGER(KIND=4), INTENT(OUT)	Vector of *n* Poisson distributed random values

Description

The vRngPoissonV function generates *n* Poisson distributed random numbers $x_i (i = 1, ..., n)$ with distribution parameter λ_i, where $\lambda_i \in R$; $\lambda_i > 0$.

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, k \in \{0, 1, 2, \ldots\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngNegBinomial
Generates random numbers with negative binomial distribution.

Syntax

status = virngnegbinomial(*method*, *stream*, *n*, *r*, *a*, *p*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
method	INTEGER, INTENT(IN)	Generation method. The specific value is: VSL_RNG_METHOD_NEGBINOMIAL_NBAR See brief description of the NBAR method in Table "Values of <*method*> in *method* parameter"
stream	TYPE (VSL_STREAM_STATE), INTENT(IN)	descriptor of the stream state structure.
n	INTEGER, INTENT(IN)	Number of random values to be generated
a	REAL(KIND=8), INTENT(IN)	The first distribution parameter *a*
p	REAL(KIND=8), INTENT(IN)	The second distribution parameter p

Output Parameters

Name	Type	Description
r	INTEGER(KIND=4), INTENT(OUT)	Vector of *n* random values with negative binomial distribution.

Description

The vRngNegBinomial function generates random numbers with negative binomial distribution and distribution parameters *a* and *p*, where *p*, $a \in R$; $0 < p < 1$; $a > 0$.

If the first distribution parameter $a \in N$, this distribution is the same as Pascal distribution. If $a \in N$, the distribution can be interpreted as the expected time of *a*-th success in a sequence of Bernoulli trials, when the probability of success is *p*.

The probability distribution is given by:

$$P(X = k) = C_{a+k-1}^{k} p^{a}(1 - p)^{k}, k \in \{0, 1, 2, \ldots\}.$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C_{a+k-1}^{k} p^{a}(1-p)^{k}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngMultinomial
Generates multinomially distributed random numbers.

Syntax

```
status = virngmultinomial( method, stream, n, r, ntrial, k, p );
```

Include Files

- mkl.fi
- mkl_vsl.f90

Input Parameters

method

INTEGER, INTENT(IN)

Generation method. The specific value is as follows:

VSL_RNG_METHOD_MULTINOMIAL_MULTPOISSON

stream

TYPE (VSL_STREAM_STATE), INTENT(IN)

Pointer to the stream state structure.

n	INTEGER, INTENT(IN) Number of random values to be generated.
ntrial	INTEGER(KIND=4), INTENT(IN) Number of independent trials *m*.
k	INTEGER(KIND=4), INTENT(IN) Number of possible outcomes.
p	REAL(KIND=8), INTENT(IN) Probability vector of k possible outcomes.

Output Parameters

r	INTEGER(KIND=4), INTENT(OUT) Array of *n* random vectors of dimension *k*.

Description

The vRngMultinomial function generates multinomially distributed random numbers with *m* independent trials and *k* possible mutually exclusive outcomes, with corresponding probabilities p_i, where $p_i \in R$; $0 \le p_i \le 1$, $m \in N$, $k \in N$.

The probability distribution is given by:

$$P(X_1 = x_1, \ldots, X_k = x_k) = \frac{m!}{\prod_{i=1}^{k} x_i!} \prod_{i=1}^{k} p_i^{x_i}, \qquad 0 \le x_i \le m, \sum_{i=1}^{k} x_i = m$$

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error (execution was successful).
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	A callback function for an abstract BRNG returns an invalid number of updated entries in a buffer; that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	A callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	An ARS-5 random number generator is not supported on the CPU running the application.

VSL_DISTR_MULTINOMIAL_BAD_PROBABILITY_ARRAY	Bad multinomial distribution probability array.

Advanced Service Routines

This section describes service routines for obtaining properties of the previously registered basic generators (vslGetBrngProperties). See VS Notes ("Basic Generators" section of VS Structure chapter) for substantiation of the need for several basic generators including user-defined BRNGs.

> **NOTE**
> The vslRegisterBrng function is provided in C for registering a user-defined basic generator, but it is not supported for Fortran. If you need to use a user-defined generator in Fortran, use an abstract basic random generator and abstract stream as described in the VS Notes.

Advanced Service Routine Data Types

The Advanced Service routines refer to a structure defining the properties of the basic generator.

This structure is described in Fortran 90 as follows:

```
TYPE VSL_BRNG_PROPERTIES
    INTEGER streamstatesize
    INTEGER nseeds
    INTEGER includeszero
    INTEGER wordsize
    INTEGER nbits
    INTEGER reserved(8)
END TYPE VSL_BRNG_PROPERTIES
```

The following table provides brief descriptions of the fields engaged in the above structure:

Field Descriptions

Field	Short Description
streamstatesize	The size, in bytes, of the stream state structure for a given basic generator.
nseeds	The number of 32-bit initial conditions (seeds) necessary to initialize the stream state structure for a given basic generator.
includeszero	Flag value indicating whether the generator can produce a random 0.
wordsize	Machine word size, in bytes, used in integer-value computations. Possible values: 4, 8, and 16 for 32, 64, and 128-bit generators, respectively.
nbits	The number of bits required to represent a random value in integer arithmetic. Note that, for instance, 48-bit random values are stored to 64-bit (8 byte) memory locations. In this case, wordsize/WordSize is equal to 8 (number of bytes used to store the random value), while nbits/NBits contains the actual number of bits occupied by the value (in this example, 48).
reserved(8)	Reserved for internal use.

NOTE
Using Advanced Service routines for defining generators is not supported for Fortran, but you can use the Fortran interface to get information about a previously-registered generator using `vslGetBrngProperties`.

vslGetBrngProperties

Returns structure with properties of a given basic generator.

Syntax

```
status = vslgetbrngproperties( brng, properties )
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
brng	`INTEGER(KIND=4), INTENT(IN)`	Number (index) of the registered basic generator; used for identification. See specific values in Table "Values of *brng* parameter". Negative values indicate the registration error.

Output Parameters

Name	Type	Description
properties	`TYPE(VSL_BRNG_PROPERTIES), INTENT(OUT)`	Pointer to the structure containing properties of the generator with number *brng*

Description

The `vslGetBrngProperties` function returns a structure with properties of a given basic generator.

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_RNG_ERROR_INVALID_BRNG_INDEX`	BRNG index is invalid.

Convolution and Correlation

Intel® oneAPI Math Kernel Library (oneMKL) VS provides a set of routines intended to perform linear convolution and correlation transformations for single and double precision real and complex data.

For correct definition of implemented operations, see the Mathematical Notation and Definitions.

The current implementation provides:

- Fourier algorithms for one-dimensional single and double precision real and complex data
- Fourier algorithms for multi-dimensional single and double precision real and complex data
- Direct algorithms for one-dimensional single and double precision real and complex data
- Direct algorithms for multi-dimensional single and double precision real and complex data

One-dimensional algorithms cover the following functions from the IBM* ESSL library:

```
SCONF, SCORF
SCOND, SCORD
SDCON, SDCOR
DDCON, DDCOR
SDDCON, SDDCOR.
```

Special wrappers are designed to simulate these ESSL functions. The wrappers are provided as sample sources:

```
${MKL}/examples/vslf/essl/vsl_wrappers
```

Additionally, you can browse the examples demonstrating the calculation of the ESSL functions through the wrappers:

```
${MKL}/examples/vslf/essl
```

The convolution and correlation API provides interfaces for Fortran 90 and C/89 languages. You can use the Fortran 90 interface with programs written in Fortran.

Intel® oneAPI Math Kernel Library (oneMKL) provides the `mkl_vsl.f90` header file. All header files are in the directory

`${MKL}/include`

See more details about the Fortran header in Random Number Generators topic.

The convolution and correlation API is implemented through task objects, or tasks. Task object is a data structure, or descriptor, which holds parameters that determine the specific convolution or correlation operation. Such parameters may be precision, type, and number of dimensions of user data, an identifier of the computation algorithm to be used, shapes of data arrays, and so on.

All the Intel® oneAPI Math Kernel Library (oneMKL) VS convolution and correlation routines process task objects in one way or another: either create a new task descriptor, change the parameter settings, compute mathematical results of the convolution or correlation using the stored parameters, or perform other operations. Accordingly, all routines are split into the following groups:

Task Constructors - routines that create a new task object descriptor and set up most common parameters.

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Execution Routines - compute results of the convolution or correlation operation over the actual input data, using the operation parameters held in the task descriptor.

Task Copy - routines used to make several copies of the task descriptor.

Task Destructors - routines that delete task objects and free the memory.

When the task is executed or copied for the first time, a special process runs which is called task commitment. During this process, consistency of task parameters is checked and the required work data are prepared. If the parameters are consistent, the task is tagged as committed successfully. The task remains committed until you edit its parameters. Hence, the task can be executed multiple times after a single commitment process. Since the task commitment process may include costly intermediate calculations such as preparation of Fourier transform of input data, launching the process only once can help speed up overall performance.

Convolution and Correlation Naming Conventions

The names of routines in the convolution and correlation API are written in lowercase (`vslsconvexec`), while the names of Fortran types and constants are written in uppercase. The names are not case-sensitive.

The names of routines have the following structure:

`vsl[`*datatype*`]{conv|corr}<base name>`

where

- `vsl` is a prefix indicating that the routine belongs to Intel® MKL Vector Statistics.
- `[datatype]` is optional. If present, the symbol specifies the type of the input and output data and can be `s` (for single precision real type), `d` (for double precision real type), `c` (for single precision complex type), or `z` (for double precision complex type).
- `Conv` or `Corr` specifies whether the routine refers to convolution or correlation task, respectively.
- `<base name>` field specifies a particular functionality that the routine is designed for, for example, `NewTask`, `DeleteTask`.

Convolution and Correlation Data Types

All convolution or correlation routines use the following types for specifying data objects:

Type	Data Object
FORTRAN 77: `INTEGER*4 task(2)` Fortran 90: `TYPE(VSL_CONV_TASK)`	Pointer to a task descriptor for convolution
FORTRAN 77: `INTEGER*4 task(2)` Fortran 90: `TYPE(VSL_CORR_TASK)`	Pointer to a task descriptor for correlation
FORTRAN 77: `REAL*4` Fortran 90: `REAL(KIND=4)`	Input/output user real data in single precision
FORTRAN 77: `REAL*8` Fortran 90: `REAL(KIND=8)`	Input/output user real data in double precision
FORTRAN 77: `COMLEX*8` Fortran 90: `COMPLEX(KIND=4)`	Input/output user complex data in single precision
FORTRAN 77: `COMPLEX*16` Fortran 90: `COMPLEX(KIND=8)`	Input/output user complex data in double precision
FORTRAN 77: `INTEGER` Fortran 90: `INTEGER`	All other data

Generic integer type (without specifying the byte size) is used for all integer data.

NOTE
The actual size of the generic integer type is platform-dependent. Before you compile your application, set an appropriate byte size for integers. See details in the 'Using the ILP64 Interface vs. LP64 Interface' section of the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Convolution and Correlation Parameters

Basic parameters held by the task descriptor are assigned values when the task object is created, copied, or modified by task editors. Parameters of the correlation or convolution task are initially set up by task constructors when the task object is created. Parameter changes or additional settings are made by task editors. More parameters which define location of the data being convolved need to be specified when the task execution routine is invoked.

According to how the parameters are passed or assigned values, all of them can be categorized as either explicit (directly passed as routine parameters when a task object is created or executed) or optional (assigned some default or implicit values during task construction).

The following table lists all applicable parameters used in the Intel® oneAPI Math Kernel Library (oneMKL) convolution and correlation API.

Convolution and Correlation Task Parameters

Name	Category	Type	Default Value Label	Description
job	explicit	integer	Implied by the constructor name	Specifies whether the task relates to convolution or correlation
type	explicit	integer	Implied by the constructor name	Specifies the type (real or complex) of the input/output data. Set to real in the current version.
precision	explicit	integer	Implied by the constructor name	Specifies precision (single or double) of the input/output data to be provided in arrays *x*,*y*,*z*.
mode	explicit	integer	None	Specifies whether the convolution/ correlation computation should be done via Fourier transforms, or by a direct method, or by automatically choosing between the two. See `SetMode` for the list of named constants for this parameter.
method	optional	integer	"auto"	Hints at a particular computation method if several methods are available for the given *mode*. Setting this parameter to "auto" means that software will choose the best available method.
internal_precision	optional	integer	Set equal to the value of *precision*	Specifies precision of internal calculations. Can enforce double precision calculations even when input/output data are single precision. See `SetInternalPrecision` for the list of named constants for this parameter.
dims	explicit	integer	None	Specifies the rank (number of dimensions) of the user data provided in arrays *x*,*y*,*z*. Can be in the range from 1 to 7.
x,*y*	explicit	real arrays	None	Specify input data arrays. See Data Allocation for more information.
z	explicit	real array	None	Specifies output data array. See Data Allocation for more information.
xshape, *yshape*, *zshape*	explicit	integer arrays	None	Define shapes of the arrays *x*, *y*, *z*. See Data Allocation for more information.
xstride, *ystride*, *zstride*	explicit	integer arrays	None	Define strides within arrays *x*, *y*, *z*, that is specify the physical location of the input and output data in these arrays. See Data Allocation for more information.

Name	Category	Type	Default Value Label	Description
start	optional	integer array	Undefined	Defines the first element of the mathematical result that will be stored to output array *z*. See SetStart and Data Allocation for more information.
decimation	optional	integer array	Undefined	Defines how to thin out the mathematical result that will be stored to output array *z*. See SetDecimation and Data Allocation for more information.

Convolution and Correlation Task Status and Error Reporting

The task status is an integer value, which is zero if no error has been detected while processing the task, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings.

An error can be caused by invalid parameter values, a system fault like a memory allocation failure, or can be an internal error self-detected by the software.

Each task descriptor contains the current status of the task. When creating a task object, the constructor assigns the VSL_STATUS_OK status to the task. When processing the task afterwards, other routines such as editors or executors can change the task status if an error occurs and write a corresponding error code into the task status field.

Note that at the stage of creating a task or editing its parameters, the set of parameters may be inconsistent. The parameter consistency check is only performed during the task commitment operation, which is implicitly invoked before task execution or task copying. If an error is detected at this stage, task execution or task copying is terminated and the task descriptor saves the corresponding error code. Once an error occurs, any further attempts to process that task descriptor is terminated and the task keeps the same error code.

Normally, every convolution or correlation function (except DeleteTask) returns the status assigned to the task while performing the function operation.

The header files define symbolic names for the status codes. These names are defined as integer constants via the PARAMETER operators.

If there is no error, the VSL_STATUS_OK status is returned, which is defined as zero:

```
F90/F95:        INTEGER(KIND=4) VSL_STATUS_OK
                PARAMETER(VSL_STATUS_OK = 0)
```

In case of an error, a non-zero error code is returned, which indicates the origin of the failure. The following status codes for the convolution/correlation error codes are pre-defined in the header files.

Convolution/Correlation Status Codes

Status Code	Description
VSL_CC_ERROR_NOT_IMPLEMENTED	Requested functionality is not implemented.
VSL_CC_ERROR_ALLOCATION_FAILURE	Memory allocation failure.
VSL_CC_ERROR_BAD_DESCRIPTOR	Task descriptor is corrupted.
VSL_CC_ERROR_SERVICE_FAILURE	A service function has failed.
VSL_CC_ERROR_EDIT_FAILURE	Failure while editing the task.
VSL_CC_ERROR_EDIT_PROHIBITED	You cannot edit this parameter.

Status Code	Description
VSL_CC_ERROR_COMMIT_FAILURE	Task commitment has failed.
VSL_CC_ERROR_COPY_FAILURE	Failure while copying the task.
VSL_CC_ERROR_DELETE_FAILURE	Failure while deleting the task.
VSL_CC_ERROR_BAD_ARGUMENT	Bad argument or task parameter.
VSL_CC_ERROR_JOB	Bad parameter: *job*.
SL_CC_ERROR_KIND	Bad parameter: *kind*.
VSL_CC_ERROR_MODE	Bad parameter: *mode*.
VSL_CC_ERROR_METHOD	Bad parameter: *method*.
VSL_CC_ERROR_TYPE	Bad parameter: *type*.
VSL_CC_ERROR_EXTERNAL_PRECISION	Bad parameter: *external_precision*.
VSL_CC_ERROR_INTERNAL_PRECISION	Bad parameter: *internal_precision*.
VSL_CC_ERROR_PRECISION	Incompatible external/internal precisions.
VSL_CC_ERROR_DIMS	Bad parameter: *dims*.
VSL_CC_ERROR_XSHAPE	Bad parameter: *xshape*.
VSL_CC_ERROR_YSHAPE	Bad parameter: *yshape*.
	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or >nmax.
VSL_CC_ERROR_ZSHAPE	Bad parameter: *zshape*.
VSL_CC_ERROR_XSTRIDE	Bad parameter: *xstride*.
VSL_CC_ERROR_YSTRIDE	Bad parameter: *ystride*.
VSL_CC_ERROR_ZSTRIDE	Bad parameter: *zstride*.
VSL_CC_ERROR_X	Bad parameter: *x*.
VSL_CC_ERROR_Y	Bad parameter: *y*.
VSL_CC_ERROR_Z	Bad parameter: *z*.
VSL_CC_ERROR_START	Bad parameter: *start*.
VSL_CC_ERROR_DECIMATION	Bad parameter: *decimation*.
VSL_CC_ERROR_OTHER	Another error.

Convolution and Correlation Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters. No additional parameter adjustment is typically required and other routines can use the task object.

Intel® MKL implementation of the convolution and correlation API provides two different forms of constructors: a general form and an X-form. X-form constructors work in the same way as the general form constructors but also assign particular data to the first operand vector used in the convolution or correlation operation (stored in array *x*).

Using X-form constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector held in array *x* against different vectors held in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Each constructor routine has an associated one-dimensional version that provides algorithmic and computational benefits.

NOTE
If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

The Table "Task Constructors" lists available task constructors:

Task Constructors

Routine	Description
vslConvNewTask/vslCorrNewTask	Creates a new convolution or correlation task descriptor for a multidimensional case.
vslConvNewTask1D/ vslCorrNewTask1D	Creates a new convolution or correlation task descriptor for a one-dimensional case.
vslConvNewTaskX/vslCorrNewTaskX	Creates a new convolution or correlation task descriptor as an X-form for a multidimensional case.
vslConvNewTaskX1D/ vslCorrNewTaskX1D	Creates a new convolution or correlation task descriptor as an X-form for a one-dimensional case.

vslConvNewTask/vslCorrNewTask

Creates a new convolution or correlation task descriptor for multidimensional case.

Syntax

```
status = vslsconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vsldconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslcconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslzconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslscorrnewtask(task, mode, dims, xshape, yshape, zshape)
status = vsldcorrnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslccorrnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslzcorrnewtask(task, mode, dims, xshape, yshape, zshape)
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
mode	FORTRAN 77: INTEGER Fortran 90: INTEGER	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
dims	FORTRAN 77: INTEGER Fortran 90: INTEGER	Rank of user data. Specifies number of dimensions for the input and output arrays *x*, *y*, and *z* used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
xshape	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION(*)	Defines the shape of the input data for the source array *x*. See Data Allocation for more information.
yshape	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION(*)	Defines the shape of the input data for the source array *y*. See Data Allocation for more information.
zshape	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION(*)	Defines the shape of the output data to be stored in array *z*. See Data Allocation for more information.

Output Parameters

Name	Type	Description
task	FORTRAN 77: INTEGER*4 task(2) for vslsconvnewtask, vsldconvnewtask, vslcconvnewtask, vslzconvnewtask INTEGER*4 task(2) for vslscorrnewtask, vsldcorrnewtask, vslccorrnewtask, vslzcorrnewtask Fortran 90: TYPE(VSL_CONV_TASK) for vslsconvnewtask, vsldconvnewtask, vslcconvnewtask, vslzconvnewtask TYPE(VSL_CORR_TASK) for vslscorrnewtask, vsldcorrnewtask, vslccorrnewtask, vslzcorrnewtask	Pointer to the task descriptor if created successfully or NULL pointer otherwise.

Name	Type	Description
status	FORTRAN 77: INTEGER Fortran 90: INTEGER	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Description

Each vslConvNewTask/vslCorrNewTask constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table "Convolution and Correlation Task Parameters").

The parameters *xshape*, *yshape*, and *zshape* define the shapes of the input and output data provided by the arrays *x*, *y*, and *z*, respectively. Each shape parameter is an array of integers with its length equal to the value of *dims*. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter *dims* is 1, then *xshape*, *yshape*, *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. Note that values of shape parameters may differ from physical shapes of arrays *x*, *y*, and *z* if non-trivial strides are assigned.

If the constructor fails to create a task descriptor, it returns a NULL task pointer.

vslConvNewTask1D/vslCorrNewTask1D

Creates a new convolution or correlation task descriptor for one-dimensional case.

Syntax

```
status = vslsconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslcconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslzconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslscorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldcorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vslccorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vslzcorrnewtask1d(task, mode, xshape, yshape, zshape)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
mode	INTEGER	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
xshape	INTEGER	Defines the length of the input data sequence for the source array *x*. See Data Allocation for more information.
yshape	INTEGER	Defines the length of the input data sequence for the source array *y*. See Data Allocation for more information.

Name	Type	Description
zshape	INTEGER	Defines the length of the output data sequence to be stored in array *z*. See Data Allocation for more information.

Output Parameters

Name	Type	Description
task	FORTRAN 77: INTEGER*4 task(2) for vslsconvnewtask1d, vsldconvnewtask1d, vslcconvnewtask1d, vslzconvnewtask1d INTEGER*4 task(2) for vslscorrnewtask1d, vsldcorrnewtask1d, vslccorrnewtask1d, vslzcorrnewtask1d TYPE(VSL_CONV_TASK) for vslsconvnewtask1d, vsldconvnewtask1d, vslcconvnewtask1d, vslzconvnewtask1d TYPE(VSL_CORR_TASK) for vslscorrnewtask1d, vsldcorrnewtask1d, vslccorrnewtask1d, vslzcorrnewtask1d VSLCorrTaskPtr* for vslsCorrNewTask1D, vsldCorrNewTask1D, vslcCorrNewTask1D, vslzCorrNewTask1D	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
status	INTEGER	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Description

Each vslConvNewTask1D/vslCorrNewTask1D constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table "Convolution and Correlation Task Parameters"). Unlike vslConvNewTask/vslCorrNewTask, these routines represent a special one-dimensional version of the constructor which assumes that the value of the parameter *dims* is 1. The parameters *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. You explicitly assign the shape parameters when calling the constructor.

vslConvNewTaskX/vslCorrNewTaskX

Creates a new convolution or correlation task descriptor for multidimensional case and assigns source data to the first operand vector.

Syntax

```
status = vslsconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vsldconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslcconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslzconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslscorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vsldcorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslccorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslzcorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
mode	INTEGER	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
dims	INTEGER	Rank of user data. Specifies number of dimensions for the input and output arrays *x*, *y*, and *z* used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
xshape	INTEGER, DIMENSION(*)	Defines the shape of the input data for the source array *x*. See Data Allocation for more information.
yshape	INTEGER, DIMENSION(*)	Defines the shape of the input data for the source array *y*. See Data Allocation for more information.
zshape	INTEGER, DIMENSION(*)	Defines the shape of the output data to be stored in array *z*.See Data Allocation for more information.
x	REAL*8 for real data in double precision flavors, COMPLEX*8 for complex data in single precision flavors, COMPLEX*16 for complex data in double precision flavors REAL(KIND=4), DIMENSION (*) for real data in single precision flavors,	Pointer to the array containing input data for the first operand vector.See Data Allocation for more information.

Name	Type	Description
	`REAL(KIND=8), DIMENSION (*)` for real data in double precision flavors, `COMPLEX(KIND=4), DIMENSION (*)` for complex data in single precision flavors, `COMPLEX(KIND=8), DIMENSION (*)` for complex data in double precision flavors	
xstride	`INTEGER, DIMENSION(*)`	Strides for input data in the array *x*.

Output Parameters

Name	Type	Description

task	`INTEGER*4 task(2)` for `vslscorrnewtaskx`, `vsldcorrnewtaskx`, `vslccorrnewtaskx`, `vslzcorrnewtaskx`
	`TYPE(VSL_CONV_TASK)` for `vslsconvnewtaskx`, `vsldconvnewtaskx`, `vslcconvnewtaskx`, `vslzconvnewtaskx`
	`TYPE(VSL_CORR_TASK)` for `vslscorrnewtaskx`, `vsldcorrnewtaskx`, `vslccorrnewtaskx`, `vslzcorrnewtaskx`
	`VSLCorrTaskPtr*` for `vslsCorrNewTaskX`, `vsldCorrNewTaskX`, `vslcCorrNewTaskX`, `vslzCorrNewTaskX`
		Pointer to the task descriptor if created successfully or `NULL` pointer otherwise.

status	`INTEGER`	Set to `VSL_STATUS_OK` if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTaskX`/`vslCorrNewTaskX` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table "Convolution and Correlation Task Parameters").

Unlike vslConvNewTask/vslCorrNewTask, these routines represent the so called X-form version of the constructor, which means that in addition to creating the task descriptor they assign particular data to the first operand vector in array *x* used in convolution or correlation operation. The task descriptor created by the vslConvNewTaskX/vslCorrNewTaskX constructor keeps the pointer to the array *x* all the time, that is, until the task object is deleted by one of the destructor routines (see vslConvDeleteTask/vslCorrDeleteTask).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters *xshape*, *yshape*, and *zshape* define the shapes of the input and output data provided by the arrays *x*, *y*, and *z*, respectively. Each shape parameter is an array of integers with its length equal to the value of *dims*. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter *dims* is 1, then *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. Note that values of shape parameters may differ from physical shapes of arrays *x*, *y*, and *z* if non-trivial strides are assigned.

The stride parameter *xstride* specifies the physical location of the input data in the array *x*. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *xstride* is s, then only every s^{th} element of the array *x* will be used to form the input sequence. The stride value must be positive or negative but not zero.

vslConvNewTaskX1D/vslCorrNewTaskX1D

Creates a new convolution or correlation task descriptor for one-dimensional case and assigns source data to the first operand vector.

Syntax

status = vslsconvnewtaskx1d(*task*, *mode*, *xshape*, *yshape*, *zshape*, *x*, *xstride*)

status = vsldconvnewtaskx1d(*task*, *mode*, *xshape*, *yshape*, *zshape*, *x*, *xstride*)

status = vslcconvnewtaskx1d(*task*, *mode*, *xshape*, *yshape*, *zshape*, *x*, *xstride*)

status = vslzconvnewtaskx1d(*task*, *mode*, *xshape*, *yshape*, *zshape*, *x*, *xstride*)

status = vslscorrnewtaskx1d(*task*, *mode*, *xshape*, *yshape*, *zshape*, *x*, *xstride*)

status = vsldcorrnewtaskx1d(*task*, *mode*, *xshape*, *yshape*, *zshape*, *x*, *xstride*)

status = vslccorrnewtaskx1d(*task*, *mode*, *xshape*, *yshape*, *zshape*, *x*, *xstride*)

status = vslzcorrnewtaskx1d(*task*, *mode*, *xshape*, *yshape*, *zshape*, *x*, *xstride*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
mode	INTEGER	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
xshape	INTEGER	Defines the length of the input data sequence for the source array *x*. See Data Allocation for more information.

Name	Type	Description
yshape	INTEGER	Defines the length of the input data sequence for the source array *y*. See Data Allocation for more information.
zshape	INTEGER	Defines the length of the output data sequence to be stored in array *z*. See Data Allocation for more information.
x	REAL*8 for real data in double precision flavors, COMPLEX*8 for complex data in single precision flavors, COMPLEX*16 for complex data in double precision flavors REAL(KIND=4), DIMENSION (*) for real data in single precision flavors, REAL(KIND=8), DIMENSION (*) for real data in double precision flavors, COMPLEX(KIND=4), DIMENSION (*) for complex data in single precision flavors, COMPLEX(KIND=8), DIMENSION (*) for complex data in double precision flavors	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.
xstride	INTEGER	Stride for input data sequence in the array*x*.

Output Parameters

Name	Type	Description
task	INTEGER*4 task(2) for vslscorrnewtaskx1d, vsldcorrnewtaskx1d, vslccorrnewtaskx1d, vslzcorrnewtaskx1d TYPE(VSL_CONV_TASK) for vslsconvnewtaskx1d, vsldconvnewtaskx1d, vslcconvnewtaskx1d, vslzconvnewtaskx1d TYPE(VSL_CORR_TASK) for vslscorrnewtaskx1d, vsldcorrnewtaskx1d, vslccorrnewtaskx1d, vslzcorrnewtaskx1d	Pointer to the task descriptor if created successfully or NULL pointer otherwise.

Name	Type	Description
	`VSLCorrTaskPtr*` for `vslsCorrNewTaskX1D`, `vsldCorrNewTaskX1D`, `vslcCorrNewTaskX1D`, `vslzCorrNewTaskX1D`	
status	`INTEGER`	Set to `VSL_STATUS_OK` if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTaskX1D`/`vslCorrNewTaskX1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table "Convolution and Correlation Task Parameters").

These routines represent a special one-dimensional version of the so called X-form of the constructor. This assumes that the value of the parameter *dims* is 1 and that in addition to creating the task descriptor, constructor routines assign particular data to the first operand vector in array *x* used in convolution or correlation operation. The task descriptor created by the `vslConvNewTaskX1D`/`vslCorrNewTaskX1D` constructor keeps the pointer to the array *x* all the time, that is, until the task object is deleted by one of the destructor routines (see `vslConvDeleteTask`/`vslCorrDeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. You explicitly assign the shape parameters when calling the constructor.

The stride parameters *xstride* specifies the physical location of the input data in the array *x* and is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *xstride* is `s`, then only every s^{th} element of the array *x* will be used to form the input sequence. The stride value must be positive or negative but not zero.

Convolution and Correlation Task Editors

Task editors in convolution and correlation API of Intel® oneAPI Math Kernel Library (oneMKL) are routines intended for setting up or changing the following task parameters (seeTable "Convolution and Correlation Task Parameters"):

- *mode*
- *internal_precision*
- *start*
- *decimation*

For setting up or changing each of the above parameters, a separate routine exists.

NOTE
Fields of the task descriptor structure are accessible only through the set of task editor routines provided with the software.

The work data computed during the last commitment process may become invalid with respect to new parameter settings. That is why after applying any of the editor routines to change the task descriptor settings, the task loses its commitment status and goes through the full commitment process again during the next execution or copy operation. For more information on task commitment, see the Introduction to Convolution and Correlation.

Table "Task Editors" lists available task editors.

Task Editors

Routine	Description
vslConvSetMode/vslCorrSetMode	Changes the value of the parameter *mode* for the operation of convolution or correlation.
vslConvSetInternalPrecision/ vslCorrSetInternalPrecision	Changes the value of the parameter *internal_precision* for the operation of convolution or correlation.
vslConvSetStart/vslCorrSetStart	Sets the value of the parameter *start* for the operation of convolution or correlation.
vslConvSetDecimation/ vslCorrSetDecimation	Sets the value of the parameter *decimation* for the operation of convolution or correlation.

NOTE
You can use the NULL task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vslConvSetMode/vslCorrSetMode

Changes the value of the parameter mode in the convolution or correlation task descriptor.

Syntax

status = vslconvsetmode(*task*, *newmode*)

status = vslcorrsetmode(*task*, *newmode*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
task	FORTRAN 77: INTEGER*4 task(2) for vslconvsetmode INTEGER*4 task(2) for vslcorrsetmode Fortran 90: TYPE(VSL_CONV_TASK) for vslconvsetmode TYPE(VSL_CORR_TASK) for vslcorrsetmode	Pointer to the task descriptor.

Name	Type	Description
newmode	FORTRAN 77: INTEGER Fortran 90: INTEGER	New value of the parameter mode.

Output Parameters

Name	Type	Description
status	FORTRAN 77: INTEGER Fortran 90: INTEGER	Current status of the task.

Description

This function is declared in mkl_vsl.f90 for the Fortran interface.

The function routine changes the value of the parameter mode for the operation of convolution or correlation. This parameter defines whether the computation should be done via Fourier transforms of the input/output data or using a direct algorithm. Initial value for mode is assigned by a task constructor.

Predefined values for the mode parameter are as follows:

Values of *mode* parameter

Value	Purpose
VSL_CONV_MODE_FFT	Compute convolution by using fast Fourier transform.
VSL_CORR_MODE_FFT	Compute correlation by using fast Fourier transform.
VSL_CONV_MODE_DIRECT	Compute convolution directly.
VSL_CORR_MODE_DIRECT	Compute correlation directly.
VSL_CONV_MODE_AUTO	Automatically choose direct or Fourier mode for convolution.
VSL_CORR_MODE_AUTO	Automatically choose direct or Fourier mode for correlation.

vslConvSetInternalPrecision/vslCorrSetInternalPrecision

Changes the value of the parameter internal_precision in the convolution or correlation task descriptor.

Syntax

```
status = vslconvsetinternalprecision(task, precision)
status = vslcorrsetinternalprecision(task, precision)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description

task	`INTEGER*4 task(2)` for `vslcorrsetinternalprecision`
	`TYPE(VSL_CONV_TASK)` for `vslconvsetinternalprecision`
	`TYPE(VSL_CORR_TASK)` for `vslcorrsetinternalprecision`
		Pointer to the task descriptor.

precision	`INTEGER`	New value of the parameter *internal_precision*.

Output Parameters

Name	Type	Description
status	`INTEGER`	Current status of the task.

Description

The `vslConvSetInternalPrecision`/`vslCorrSetInternalPrecision` routine changes the value of the parameter *internal_precision* for the operation of convolution or correlation. This parameter defines whether the internal computations of the convolution or correlation result should be done in single or double precision. Initial value for *internal_precision* is assigned by a task constructor and set to either "single" or "double" according to the particular flavor of the constructor used.

Changing the *internal_precision* can be useful if the default setting of this parameter was "single" but you want to calculate the result with double precision even if input and output data are represented in single precision.

Predefined values for the *internal_precision* input parameter are as follows:

Values of *internal_precision* Parameter

Value	Purpose
`VSL_CONV_PRECISION_SINGLE`	Compute convolution with single precision.
`VSL_CORR_PRECISION_SINGLE`	Compute correlation with single precision.
`VSL_CONV_PRECISION_DOUBLE`	Compute convolution with double precision.
`VSL_CORR_PRECISION_DOUBLE`	Compute correlation with double precision.

vslConvSetStart/vslCorrSetStart

Changes the value of the parameter start in the convolution or correlation task descriptor.

Syntax

```
status = vslconvsetstart(task, start)
status = vslcorrsetstart(task, start)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
task	INTEGER*4 task(2) for vslcorrsetstart TYPE(VSL_CONV_TASK) for vslconvsetstart TYPE(VSL_CORR_TASK) for vslcorrsetstart	Pointer to the task descriptor.
start	INTEGER, DIMENSION (*)	New value of the parameter *start*.

Output Parameters

Name	Type	Description
status	INTEGER	Current status of the task.

Description

The vslConvSetStart/vslCorrSetStart routine sets the value of the parameter *start* for the operation of convolution or correlation. In a one-dimensional case, this parameter points to the first element in the mathematical result that should be stored in the output array. In a multidimensional case, *start* is an array of indices and its length is equal to the number of dimensions specified by the parameter *dims*. For more information about the definition and effect of this parameter, see Data Allocation.

During the initial task descriptor construction, the default value for *start* is undefined and this parameter is not used. Therefore the only way to set and use the *start* parameter is via assigning it some value by one of the vslConvSetStart/vslCorrSetStart routines.

vslConvSetDecimation/vslCorrSetDecimation

Changes the value of the parameter decimation in the convolution or correlation task descriptor.

Syntax

status = vslconvsetdecimation(*task*, *decimation*)

status = vslcorrsetdecimation(*task*, *decimation*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
task	INTEGER*4 task(2) for vslcorrsetdecimation	Pointer to the task descriptor.

Name	Type	Description
	`TYPE(VSL_CONV_TASK)` for `vslconvsetdecimation` `TYPE(VSL_CORR_TASK)` for `vslcorrsetdecimation`	
decimation	`INTEGER, DIMENSION (*)`	New value of the parameter *decimation*.

Output Parameters

Name	Type	Description
status	`INTEGER`	Current status of the task.

Description

The routine sets the value of the parameter *decimation* for the operation of convolution or correlation. This parameter determines how to thin out the mathematical result of convolution or correlation before writing it into the output data array. For example, in a one-dimensional case, if *decimation* = *d* > 1, only every *d*-th element of the mathematical result is written to the output array *z*. In a multidimensional case, *decimation* is an array of indices and its length is equal to the number of dimensions specified by the parameter *dims*. For more information about the definition and effect of this parameter, see Data Allocation.

During the initial task descriptor construction, the default value for *decimation* is undefined and this parameter is not used. Therefore the only way to set and use the *decimation* parameter is via assigning it some value by one of the `vslSetDecimation` routines.

Task Execution Routines

Task execution routines compute convolution or correlation results based on parameters held by the task descriptor and on the user data supplied for input vectors.

After you create and adjust a task, you can execute it multiple times by applying to different input/output data of the same type, precision, and shape.

Intel® oneAPI Math Kernel Library (oneMKL) provides the following forms of convolution/correlation execution routines:

- **General form** executors that use the task descriptor created by the general form constructor and expect to get two source data arrays *x* and *y* on input
- **X-form** executors that use the task descriptor created by the X-form constructor and expect to get only one source data array *y* on input because the first array *x* has been already specified on the construction stage

When the task is executed for the first time, the execution routine includes a task commitment operation, which involves two basic steps: parameters consistency check and preparation of auxiliary data (for example, this might be the calculation of Fourier transform for input data).

Each execution routine has an associated one-dimensional version that provides algorithmic and computational benefits.

NOTE
You can use the `NULL` task pointer in calls to execution routines. In this case, the routine is terminated and no system crash occurs.

If the task is executed successfully, the execution routine returns the zero status code. If an error is detected, the execution routine returns an error code which signals that a specific error has occurred. In particular, an error status code is returned in the following cases:

- if the task pointer is `NULL`
- if the task descriptor is corrupted
- if calculation has failed for some other reason.

> **NOTE**
> Intel® MKL does not control floating-point errors, like overflow or gradual underflow, or operations with NaNs, etc.

If an error occurs, the task descriptor stores the error code.

The table below lists all task execution routines.

Task Execution Routines

Routine	Description
vslConvExec/vslCorrExec	Computes convolution or correlation for a multidimensional case.
vslConvExec1D/vslCorrExec1D	Computes convolution or correlation for a one-dimensional case.
vslConvExecX/vslCorrExecX	Computes convolution or correlation as X-form for a multidimensional case.
vslConvExecX1D/vslCorrExecX1D	Computes convolution or correlation as X-form for a one-dimensional case.

vslConvExec/vslCorrExec

Computes convolution or correlation for multidimensional case.

Syntax

```
status = vslsconvexec(task, x, xstride, y, ystride, z, zstride)
status = vsldconvexec(task, x, xstride, y, ystride, z, zstride)
status = vslcconvexec(task, x, xstride, y, ystride, z, zstride)
status = vslzconvexec(task, x, xstride, y, ystride, z, zstride)
status = vslscorrexec(task, x, xstride, y, ystride, z, zstride)
status = vsldcorrexec(task, x, xstride, y, ystride, z, zstride)
status = vslccorrexec(task, x, xstride, y, ystride, z, zstride)
status = vslzcorrexec(task, x, xstride, y, ystride, z, zstride)
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
task	`INTEGER*4 task(2)` for `vslscorrexec`, `vsldcorrexec`, `vslccorrexec`, `vslzcorrexec`	Pointer to the task descriptor

Name	Type	Description
	`TYPE(VSL_CONV_TASK)` for `vslsconvexec`, `vsldconvexec`, `vslcconvexec`, `vslzconvexec`	
	`TYPE(VSL_CORR_TASK)` for `vslscorrexec`, `vsldcorrexec`, `vslccorrexec`, `vslzcorrexec`	
	`VSLCorrTaskPtr` for `vslsCorrExec`, `vsldCorrExec`, `vslcCorrExec`, `vslzCorrExec`	
x, *y*	`REAL*8` for `vsldconvexec` and `vsldcorrexec`,	Pointers to arrays containing input data. See Data Allocation for more information.
	`COMPLEX*8` for`vslcconvexec` and `vslccorrexec`,	
	`COMPLEX*16` for`vslzconvexec` and `vslzcorrexec`	
	`REAL(KIND=4), DIMENSION(*)` for `vslsconvexec` and `vslscorrexec`,	
	`REAL(KIND=8), DIMENSION(*)` for `vsldconvexec` and `vsldcorrexec`,	
	`COMPLEX(KIND=4), DIMENSION(*)` for`vslcconvexec` and `vslccorrexec`,	
	`COMPLEX(KIND=8), DIMENSION(*)` for `vslzconvexec` and `vslzcorrexec`	
xstride, *ystride*, *zstride*	`INTEGER, DIMENSION (*)`	Strides for input and output data. For more information, see stride parameters.

Output Parameters

Name	Type	Description
z	`REAL*8` for `vsldconvexec` and `vsldcorrexec`,	Pointer to the array that stores output data. See Data Allocation for more information.
	`COMPLEX*8` for`vslcconvexec` and `vslccorrexec`,	
	`COMPLEX*16` for`vslzconvexec` and `vslzcorrexec`	
	`REAL(KIND=4), DIMENSION(*)` for `vslsconvexec` and `vslscorrexec`,	

Name	Type	Description
	`REAL(KIND=8), DIMENSION(*)` for `vsldconvexec` and `vsldcorrexec`, `COMPLEX(KIND=4), DIMENSION(*)` for`vslcconvexec` and `vslccorrexec`, `COMPLEX(KIND=8), DIMENSION(*)` for `vslzconvexec` and `vslzcorrexec`	
status	INTEGER	Set to `VSL_STATUS_OK` if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExec`/`vslCorrExec` routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTask`/`vslCorrNewTask` constructor and pointed to by *task*. If *task* is `NULL`, no operation is done.

The stride parameters *xstride*, *ystride*, and *zstride* specify the physical location of the input and output data in the arrays *x*, *y*, and *z*, respectively. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *zstride* is `s`, then only every s^{th} element of the array *z* will be used to store the output data. The stride value must be positive or negative but not zero.

vslConvExec1D/vslCorrExec1D

Computes convolution or correlation for one-dimensional case.

Syntax

```
status = vslsconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vsldconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslcconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslzconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslscorrexec1d(task, x, xstride, y, ystride, z, zstride)
status = vsldcorrexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslccorrexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslzcorrexec1d(task, x, xstride, y, ystride, z, zstride)
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description

task	`INTEGER*4 task(2)` for `vslscorrexec1d`, `vsldcorrexec1d`, `vslccorrexec1d`, `vslzcorrexec1d`
	`TYPE(VSL_CONV_TASK)` for `vslsconvexec1d`, `vsldconvexec1d`, `vslcconvexec1d`, `vslzconvexec1d`
	`TYPE(VSL_CORR_TASK)` for `vslscorrexec1d`, `vsldcorrexec1d`, `vslccorrexec1d`, `vslzcorrexec1d`
	`VSLCorrTaskPtr` for `vslsCorrExec1D`, `vsldCorrExec1D`, `vslcCorrExec1D`, `vslzCorrExec1D`
		Pointer to the task descriptor.

x, *y*	`REAL*8` for `vsldconvexec1d` and `vsldcorrexec1d`,
	`COMPLEX*8` for`vslcconvexec1d` and `vslccorrexec1d`,
	`COMPLEX*16` for`vslzconvexec1d` and `vslzcorrexec1d`
	`REAL(KIND=4), DIMENSION(*)` for `vslsconvexec1d` and `vslscorrexec1d`,
	`REAL(KIND=8), DIMENSION(*)` for `vsldconvexec1d` and `vsldcorrexec1d`,
	`COMPLEX(KIND=4), DIMENSION(*)` for`vslcconvexec1d` and `vslccorrexec1d`,
	`COMPLEX(KIND=8), DIMENSION(*)` for `vslzconvexec1d` and `vslzcorrexec1d`
		Pointers to arrays containing input data. See Data Allocation for more information.

xstride, *ystride*, *zstride*	`INTEGER`	Strides for input and output data. For more information, see stride parameters.

Output Parameters

Name	Type	Description
z	REAL*8 for vsldconvexec1d and vsldcorrexec1d, COMPLEX*8 forvslcconvexec1d and vslccorrexec1d, COMPLEX*16 forvslzconvexec1d and vslzcorrexec1d REAL(KIND=4), DIMENSION(*) for vslsconvexec1d and vslscorrexec1d, REAL(KIND=8), DIMENSION(*) for vsldconvexec1d and vsldcorrexec1d, COMPLEX(KIND=4), DIMENSION(*) forvslcconvexec1d and vslccorrexec1d, COMPLEX(KIND=8), DIMENSION(*) for vslzconvexec1d and vslzcorrexec1d	Pointer to the array that stores output data. See Data Allocation for more information.
status	INTEGER	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the vslConvExec1D/vslCorrExec1D routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. These routines represent a special one-dimensional version of the operation, assuming that the value of the parameter *dims* is 1. Using this version of execution routines can help speed up performance in case of one-dimensional data.

Parameters of the operation are read from the task descriptor created previously by a corresponding vslConvNewTask1D/vslCorrNewTask1D constructor and pointed to by *task*. If *task* is NULL, no operation is done.

vslConvExecX/vslCorrExecX

Computes convolution or correlation for multidimensional case with the fixed first operand vector.

Syntax

```
status = vslsconvexecx(task, y, ystride, z, zstride)
status = vsldconvexecx(task, y, ystride, z, zstride)
status = vslcconvexecx(task, y, ystride, z, zstride)
status = vslzconvexecx(task, y, ystride, z, zstride)
status = vslscorrexecx(task, y, ystride, z, zstride)
```

```
status = vsldcorrexecx(task, y, ystride, z, zstride)
status = vslccorrexecx(task, y, ystride, z, zstride)
status = vslzcorrexecx(task, y, ystride, z, zstride)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
task	INTEGER*4 task(2) for vslscorrexecx, vsldcorrexecx, vslccorrexecx, vslzcorrexecx TYPE(VSL_CONV_TASK) for vslsconvexecx, vsldconvexecx, vslcconvexecx, vslzconvexecx TYPE(VSL_CORR_TASK) for vslscorrexecx, vsldcorrexecx, vslccorrexecx, vslzcorrexecx VSLCorrTaskPtr for vslsCorrExecX, vsldCorrExecX, vslcCorrExecX, vslzCorrExecX	Pointer to the task descriptor.
x ,*y*	REAL*8 for vsldconvexecx and vsldcorrexecx, COMPLEX*8 forvslcconvexecx and vslccorrexecx, COMPLEX*16 forvslzconvexecx and vslzcorrexecx REAL(KIND=4), DIMENSION(*) for vslsconvexecx and vslscorrexecx, REAL(KIND=8), DIMENSION(*) for vsldconvexecx and vsldcorrexecx, COMPLEX(KIND=4), DIMENSION (*) forvslcconvexecx and vslccorrexecx,	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

Name	Type	Description
	COMPLEX(KIND=8), DIMENSION (*) for vslzconvexecx and vslzcorrexecx	
ystride ,*z* *stride*	INTEGER, DIMENSION (*)	Strides for input and output data. For more information, see stride parameters.

Output Parameters

Name	Type	Description
z	REAL*8 for vsldconvexecx and vsldcorrexecx, COMPLEX*8 forvslcconvexecx and vslccorrexecx, COMPLEX*16 forvslzconvexecx and vslzcorrexecx REAL(KIND=4), DIMENSION(*) for vslsconvexecx and vslscorrexecx, REAL(KIND=8), DIMENSION(*) for vsldconvexecx and vsldcorrexecx, COMPLEX(KIND=4), DIMENSION (*) forvslcconvexecx and vslccorrexecx, COMPLEX(KIND=8), DIMENSION (*) for vslzconvexecx and vslzcorrexecx	Pointer to the array that stores output data. See Data Allocation for more information.
status	INTEGER	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the vslConvExecX/vslCorrExecX routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. These routines represent a special version of the operation, which assumes that the first operand vector was set on the task construction stage and the task object keeps the pointer to the array *x*.

Parameters of the operation are read from the task descriptor created previously by a corresponding vslConvNewTaskX/vslCorrNewTaskX constructor and pointed to by *task*. If *task* is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

vslConvExecX1D/vslCorrExecX1D

Computes convolution or correlation for one-dimensional case with the fixed first operand vector.

Syntax

```
status = vslsconvexecx1d(task, y, ystride, z, zstride)
status = vsldconvexecx1d(task, y, ystride, z, zstride)
status = vslcconvexecx1d(task, y, ystride, z, zstride)
status = vslzconvexecx1d(task, y, ystride, z, zstride)
status = vslscorrexecx1d(task, y, ystride, z, zstride)
status = vsldcorrexecx1d(task, y, ystride, z, zstride)
status = vslccorrexecx1d(task, y, ystride, z, zstride)
status = vslzcorrexecx1d(task, y, ystride, z, zstride)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
task	INTEGER*4 task(2) for vslscorrexecx1d, vsldcorrexecx1d, vslccorrexecx1d, vslzcorrexecx1d TYPE(VSL_CONV_TASK) for vslsconvexecx1d, vsldconvexecx1d, vslcconvexecx1d, vslzconvexecx1d TYPE(VSL_CORR_TASK) for vslscorrexecx1d, vsldcorrexecx1d, vslccorrexecx1d, vslzcorrexecx1d VSLCorrTaskPtr for vslsCorrExecX1D, vsldCorrExecX1D, vslcCorrExecX1D, vslzCorrExecX1D	Pointer to the task descriptor.
x, *y*	REAL*8 for vsldconvexecx1d and vsldcorrexecx1d, COMPLEX*8 forvslcconvexecx1d and vslccorrexecx1d,	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

Name	Type	Description
	COMPLEX*16 forvslzconvexecx1d and vslzcorrexecx1d REAL(KIND=4), DIMENSION(*) for vslsconvexecx1d and vslscorrexecx1d, REAL(KIND=8), DIMENSION(*) for vsldconvexecx1d and vsldcorrexecx1d, COMPLEX(KIND=4), DIMENSION(*) forvslcconvexecx1d and vslccorrexecx1d, COMPLEX(KIND=8), DIMENSION(*) for vslzconvexecx1d and vslzcorrexecx1d	
ystride, *zstride*	INTEGER	Strides for input and output data. For more information, see stride parameters.

Output Parameters

Name	Type	Description
z	REAL*8 for vsldconvexecx1d and vsldcorrexecx1d, COMPLEX*8 forvslcconvexecx1d and vslccorrexecx1d, COMPLEX*16 forvslzconvexecx1d and vslzcorrexecx1d REAL(KIND=4), DIMENSION(*) for vslsconvexecx1d and vslscorrexecx1d, REAL(KIND=8), DIMENSION(*) for vsldconvexecx1d and vsldcorrexecx1d, COMPLEX(KIND=4), DIMENSION(*) forvslcconvexecx1d and vslccorrexecx1d, COMPLEX(KIND=8), DIMENSION(*) for vslzconvexecx1d and vslzcorrexecx1d	Pointer to the array that stores output data. See Data Allocation for more information.
status	INTEGER	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the vslConvExecX1D/vslCorrExecX1D routines computes convolution or correlation of one-dimensional (assuming that *dims* =1) data provided by the arrays *x* and *y* and then stores the results in the array *z*. These routines represent a special version of the operation, which expects that the first operand vector was set on the task construction stage.

Parameters of the operation are read from the task descriptor created previously by a corresponding vslConvNewTaskX1D/vslCorrNewTaskX1D constructor and pointed to by *task*. If *task* is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple one-dimensional convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Convolution and Correlation Task Destructors

Task destructors are routines designed for deleting task objects and deallocating memory.

vslConvDeleteTask/vslCorrDeleteTask

Destroys the task object and frees the memory.

Syntax

```
errcode = vslconvdeletetask(task)
errcode = vslcorrdeletetask(task)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
task	INTEGER*4 task(2) for vslcorrdeletetask TYPE(VSL_CONV_TASK) for vslconvdeletetask TYPE(VSL_CORR_TASK) for vslcorrdeletetask	Pointer to the task descriptor.

Output Parameters

Name	Type	Description
errcode	INTEGER	Contains 0 if the task object is deleted successfully. Contains an error code if an error occurred.

Description

The vslConvDeleteTask/vslCorrvDeleteTask routine deletes the task descriptor object and frees any working memory and the memory allocated for the data structure. The task pointer is set to NULL.

Note that if the vslConvDeleteTask/vslCorrvDeleteTask routine does not delete the task successfully, the routine returns an error code. This error code has no relation to the task status code and does not change it.

NOTE
You can use the NULL task pointer in calls to destructor routines. In this case, the routine terminates with no system crash.

Convolution and Correlation Task Copiers

The routines are designed for copying convolution and correlation task descriptors.

vslConvCopyTask/vslCorrCopyTask

Copies a descriptor for convolution or correlation task.

Syntax

```
status = vslconvcopytask(newtask, srctask)
status = vslcorrcopytask(newtask, srctask)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
srctask	INTEGER*4 srctask(2) for vslcorrcopytask TYPE(VSL_CONV_TASK) for vslconvcopytask TYPE(VSL_CORR_TASK) for vslcorrcopytask	Pointer to the source task descriptor.

Output Parameters

Name	Type	Description
newtask	INTEGER*4 srctask(2) for vslcorrcopytask TYPE(VSL_CONV_TASK) for vslconvcopytask TYPE(VSL_CORR_TASK) for vslcorrcopytask	Pointer to the new task descriptor.
status	INTEGER	Current status of the source task.

Description

If a task object *srctask* already exists, you can use an appropriate vslConvCopyTask/vslCorrCopyTask routine to make its copy in *newtask*. After the copy operation, both source and new task objects will become committed (see Introduction to Convolution and Correlation for information about task commitment). If the

source task was not previously committed, the commitment operation for this task is implicitly invoked before copying starts. If an error occurs during source task commitment, the task stores the error code in the status field. If an error occurs during copy operation, the routine returns a `NULL` pointer instead of a reference to a new task object.

Convolution and Correlation Usage Examples

This section demonstrates how you can use the Intel® oneAPI Math Kernel Library (oneMKL) routines to perform some common convolution and correlation operations both for single-threaded and multithreaded calculations. The following two sample functions`scond1` and `sconf1` simulate the convolution and correlation functions `SCOND` and `SCONF` found in IBM ESSL* library. The functions assume single-threaded calculations and can be used with C or C++ compilers.

Function scond1 for Single-Threaded Calculations

```
#include "mkl_vsl.h"

int scond1(
    float h[], int inch,
    float x[], int incx,
    float y[], int incy,
    int nh, int nx, int iy0, int ny)
{
    int status;
    VSLConvTaskPtr task;
    vslsConvNewTask1D(&task,VSL_CONV_MODE_DIRECT,nh,nx,ny);
    vslConvSetStart(task, &iy0);
    status = vslsConvExec1D(task, h,inch, x,incx, y,incy);
    vslConvDeleteTask(&task);
    return status;
}
```

Function sconf1 for Single-Threaded Calculations

```
#include "mkl_vsl.h"
int sconf1(
    int init,
    float h[], int inc1h,
    float x[], int inc1x, int inc2x,
    float y[], int inc1y, int inc2y,
    int nh, int nx, int m, int iy0, int ny,
    void* aux1, int naux1, void* aux2, int naux2)
{
    int status;
    /* assume that aux1!=0 and naux1 is big enough */
    VSLConvTaskPtr* task = (VSLConvTaskPtr*)aux1;
    if (init != 0)
        /* initialization: */
        status = vslsConvNewTaskX1D(task,VSL_CONV_MODE_FFT,
         nh,nx,ny, h,inc1h);
    if (init == 0) {
        /* calculations: */
        int i;
        vslConvSetStart(*task, &iy0);
        for (i=0; i<m; i++) {
         float* xi = &x[inc2x * i];
         float* yi = &y[inc2y * i];
         /* task is implicitly committed at i==0 */
         status = vslsConvExecX1D(*task, xi, inc1x, yi, inc1y);
        };
    };
    vslConvDeleteTask(task);
    return status;
}
```

Using Multiple Threads

For functions such as `sconf1` described in the previous example, parallel calculations may be more preferable instead of cycling. If `m>1`, you can use multiple threads for invoking the task execution against different data sequences. For such cases, use task copy routines to create `m` copies of the task object before the calculations stage and then run these copies with different threads. Ensure that you make all necessary parameter adjustments for the task (using Task Editors) before copying it.

The sample code in this case may look as follows:

```
if (init == 0) {
    int i, status, ss[M];
    VSLConvTaskPtr tasks[M];
    /* assume that M is big enough */
    . . .
    vslConvSetStart(*task, &iy0);
    . . .
    for (i=0; i<m; i++)
        /* implicit commitment at i==0 */
        vslConvCopyTask(&tasks[i],*task);
    . . .
```

Then, `m` threads may be started to execute different copies of the task:

```
. . .
        float* xi = &x[inc2x * i];
        float* yi = &y[inc2y * i];
        ss[i]=vslsConvExecX1D(tasks[i], xi,inc1x, yi,inc1y);
    . . .
```

And finally, after all threads have finished the calculations, overall status should be collected from all task objects. The following code signals the first error found, if any:

```
    . . .
    for (i=0; i<m; i++) {
        status = ss[i];
        if (status != 0) /* 0 means "OK" */
            break;
    };
    return status;
}; /* end if init==0 */
```

Execution routines modify the task internal state (fields of the task structure). Such modifications may conflict with each other if different threads work with the same task object simultaneously. That is why different threads must use different copies of the task.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Convolution and Correlation Mathematical Notation and Definitions

The following notation is necessary to explain the underlying mathematical definitions used in the text:

$\mathbf{R} = (-\infty, +\infty)$	The set of real numbers.
$\mathbf{Z} = \{0, \pm1, \pm2, \ldots\}$	The set of integer numbers.
$\mathbf{Z}^N = \mathbf{Z} \times \ldots \times \mathbf{Z}$	The set of N-dimensional series of integer numbers.
$p = (p_1, \ldots, p_N) \in \mathbf{Z}^N$	N-dimensional series of integers.
$u:\mathbf{Z}^N \to \mathbf{R}$	Function u with arguments from $\mathbf{Z}^N$ and values from $\mathbf{R}$.
$u(p) = u(p_1, \ldots, p_N)$	The value of the function u for the argument $(p_1, \ldots, p_N)$.
$w = u*v$	Function w is the convolution of the functions u, v.
$w = u \bullet v$	Function w is the correlation of the functions u, v.

Given series $p, q \in \mathbf{Z}^N$:

- series $r = p + q$ is defined as $r^n = p^n + q^n$ for every $n=1,\ldots,N$
- series $r = p - q$ is defined as $r^n = p^n - q^n$ for every $n=1,\ldots,N$
- series $r = \sup\{p, q\}$ is defines as $r^n = \max\{p^n, q^n\}$ for every $n=1,\ldots,N$
- series $r = \inf\{p, q\}$ is defined as $r^n = \min\{p^n, q^n\}$ for every $n=1,\ldots,N$
- inequality $p \le q$ means that $p^n \le q^n$ for every $n=1,\ldots,N$.

A function $u(p)$ is called a finite function if there exist series $P^{min}, P^{max} \in \mathbf{Z}^N$ such that:

$$u(p) \neq 0$$

implies

$$P^{min} \le p \le P^{max}.$$

Operations of convolution and correlation are only defined for finite functions.

Consider functions u, v and series P^{min}, $P^{max}Q^{min}$, $Q^{max} \in \mathbf{Z}^N$ such that:

$u(p) \neq 0$ implies $P^{min} \le p \le P^{max}$.
$v(q) \neq 0$ implies $Q^{min} \le q \le Q^{max}$.

Definitions of linear correlation and linear convolution for functions u and v are given below.

Linear Convolution

If function $w = u*v$ is the convolution of u and v, then:

$w(r) \neq 0$ implies $\mathbf{R}^{min} \le r \le \mathbf{R}^{max}$,
where $\mathbf{R}^{min} = P^{min} + Q^{min}$ and $\mathbf{R}^{max} = P^{max} + Q^{max}$.

If $\mathbf{R}^{min} \le r \le \mathbf{R}^{max}$, then:

$w(r) = \sum u(t) \cdot v(r-t)$ is the sum for all $t \in \mathbf{Z}^N$ such that $\mathbf{T}^{min} \le t \le \mathbf{T}^{max}$,
where $\mathbf{T}^{min} = \sup\{P^{min}, r - Q^{max}\}$ and $\mathbf{T}^{max} = \inf\{P^{max}, r - Q^{min}\}$.

Linear Correlation

If function $w = u \bullet v$ is the correlation of u and v, then:

$w(r) \neq 0$ implies $\mathbf{R}^{min} \le r \le \mathbf{R}^{max}$,
where $\mathbf{R}^{min} = Q^{min} - P^{max}$ and $\mathbf{R}^{max} = Q^{max} - P^{min}$.

If $\mathbf{R}^{min} \le r \le \mathbf{R}^{max}$, then:

$w(r) = \sum u(t) \cdot v(r+t)$ is the sum for all $t \in \mathbf{Z}^N$ such that $\mathbf{T}^{min} \le t \le \mathbf{T}^{max}$,
where $\mathbf{T}^{min} = \sup\{P^{min}, Q^{min} - r\}$ and $\mathbf{T}^{max} = \inf\{P^{max}, Q^{max} - r\}$.

Representation of the functions *u*, *v*, *w*as the input/output data for the Intel® oneAPI Math Kernel Library (oneMKL) convolution and correlation functions is described in theData Allocation.

Convolution and Correlation Data Allocation

This section explains the relation between:

- mathematical finite functions *u*, *v*, *w* introduced in Mathematical Notation and Definitions;
- multi-dimensional input and output data vectors representing the functions *u*, *v*, *w*;
- arrays *u*, *v*, *w* used to store the input and output data vectors in computer memory

The convolution and correlation routine parameters that determine the allocation of input and output data are the following:

- Data arrays *x*, *y*, *z*
- Shape arrays *xshape*, *yshape*, *zshape*
- Strides within arrays *xstride*, *ystride*, *zstride*
- Parameters *start*, *decimation*

Finite Functions and Data Vectors

The finite functions *u(p)*, *v(q)*, and *w(r)* introduced above are represented as multi-dimensional vectors of input and output data:

$inputu(i_1,\ldots,i_{dims})$ for $u(p_1,\ldots,p_N)$

$inputv(j_1,\ldots,j_{dims})$ for $v(q_1,\ldots,q_N)$

$output(k_1,\ldots,k_{dims})$ for $w(r_1,\ldots,r_N)$.

Parameter *dims* represents the number of dimensions and is equal to N.

The parameters *xshape*, *yshape*, and *zshape* define the shapes of input/output vectors:

$inputu(i_1,\ldots,i_{dims})$ is defined if $1 \le i_n \le xshape(n)$ for every $n=1,\ldots,dims$

$inputv(j_1,\ldots,j_{dims})$ is defined if $1 \le j_n \le yshape(n)$ for every $n=1,\ldots,dims$

$output(k_1,\ldots,k_{dims})$ is defined if $1 \le k_n \le zshape(n)$ for every $n=1,\ldots,dims$.

Relation between the input vectors and the functions *u* and *v* is defined by the following formulas:

$inputu(i_1,\ldots,i_{dims}) = u(p_1,\ldots,p_N)$, where $p_n = P_n^{min} + (i_n-1)$ for every n

$inputv(j_1,\ldots,j_{dims}) = v(q_1,\ldots,q_N)$, where $q_n=Q_n^{min} + (j_n-1)$ for every n.

The relation between the output vector and the function *w(r)* is similar (but only in the case when parameters *start* and *decimation* are not defined):

$output(k_1,\ldots,k_{dims}) = w(r_1,\ldots,r_N)$, where $r_n=R_n^{min} + (k_n-1)$ for every n.

If the parameter *start* is defined, it must belong to the interval $R_n^{min} \le start(n) \le R_n^{max}$. If defined, the *start* parameter replaces R^{min} in the formula:

$output(k_1,\ldots,k_{dims})=w(r_1,\ldots,r_N)$, where $r_n=start(n) + (k_n-1)$

If the parameter *decimation* is defined, it changes the relation according to the following formula:

$output(k_1,\ldots,k_{dims})=w(r_1,\ldots,r_N)$, where $r_n= R_n^{min} + (k_n-1)*decimation(n)$

If both parameters *start* and *decimation* are defined, the formula is as follows:

output(k_1,...,k_{dims})=w(r_1,...,r_N), where r_n=start(n) + (k_n-1)*decimation(n)

The convolution and correlation software checks the values of *zshape*, *start*, and *decimation* during task commitment. If r_n exceeds R_n^{max} for some k_n,n=1,...,dims, an error is raised.

Allocation of Data Vectors

Both parameter arrays *x* and *y* contain input data vectors in memory, while array *z* is intended for storing output data vector. To access the memory, the convolution and correlation software uses only pointers to these arrays and ignores the array shapes.

For parameters *x*, *y*, and *z*, you can provide one-dimensional arrays with the requirement that actual length of these arrays be sufficient to store the data vectors.

The allocation of the input and output data inside the arrays *x*, *y*, and *z* is described below assuming that the arrays are one-dimensional. Given multi-dimensional indices $i, j, k \in \mathbf{Z}^N$, one-dimensional indices $e, f, g \in \mathbf{Z}$ are defined such that:

inputu(i_1,...,i_{dims}) is allocated at *x(e)*

inputv(j_1,...,j_{dims}) is allocated at *y(f)*

output(k_1,...,k_{dims}) is allocated at *z(g)*.

The indices *e*, *f*, and *g* are defined as follows:

$e = 1 + \sum xstride(n) \cdot dx(n)$ (the sum is for all n=1,...,dims)

$f = 1 + \sum ystride(n) \cdot dy(n)$ (the sum is for all n=1,...,dims)

$g = 1 + \sum zstride(n) \cdot dz(n)$ (the sum is for all n=1,...,dims)

The distances dx(n), dy(n), and dz(n) depend on the signum of the stride:

dx(n) = i_n-1 if xstride(n)>0, or dx(n) = i_n-xshape(n) if xstride(n)<0

dy(n) = j_n-1 if ystride(n)>0, or dy(n) = j_n-yshape(n) if ystride(n)<0

dz(n) = k_n-1 if zstride(n)>0, or dz(n) = k_n-zshape(n) if zstride(n)<0

The definitions of indices *e*, *f*, and *g* assume that indexes for arrays *x*, *y*, and *z* are started from unity:

x(e) is defined for e=1,...,length(x)

y(f) is defined for f=1,...,length(y)

z(g) is defined for g=1,...,length(z)

Below is a detailed explanation about how elements of the multi-dimensional output vector are stored in the array *z* for one-dimensional and two-dimensional cases.

One-dimensional case. If *dims*=1, then *zshape* is the number of the output values to be stored in the array *z*. The actual length of array *z* may be greater than *zshape* elements.

If zstride>1, output values are stored with the stride: output(1) is stored to z(1), output(2) is stored to z(1+zstride), and so on. Hence, the actual length of *z* must be at least 1+zstride*(zshape-1) elements or more.

If zstride<0, it still defines the stride between elements of array *z*. However, the order of the used elements is the opposite. For the *k*-th output value, output(k) is stored in z(1+|zstride|*(zshape-k)), where |zstride| is the absolute value of zstride. The actual length of the array *z* must be at least 1+|zstride| *(zshape - 1) elements.

Two-dimensional case. If *dims*=2, the output data is a two-dimensional matrix. The value zstride(1) defines the stride inside matrix columns, that is, the stride between the output(k_1, k_2) and output(k_1+1, k_2) for every pair of indices k_1, k_2. On the other hand, zstride(2) defines the stride between columns, that is, the stride between output(k_1,k_2) and output(k_1,k_2+1).

If zstride(2) is greater than zshape(1), this causes sparse allocation of columns. If the value of zstride(2) is smaller than zshape(1), this may result in the transposition of the output matrix. For example, if zshape = (2,3), you can define zstride = (3,1) to allocate output values like transposed matrix of the shape 3x2.

Whether zstride assumes this kind of transformations or not, you need to ensure that different elements output (k_1, ...,k_{dims}) will be stored in different locations *z(g)*.

Summary Statistics

The Summary Statistics domain provides routines that compute basic statistical estimates for single and double precision multi-dimensional datasets.

The Summary Statistics routines calculate:

- raw and central moments up to the fourth order
- skewness and excess kurtosis (further referred to as *kurtosis* for brevity)
- variation coefficient
- quantiles and order statistics
- minimum and maximum
- variance-covariance/correlation matrix
- pooled/group variance-covariance matrix and mean
- partial variance-covariance/correlation matrix
- robust estimators for variance-covariance matrix and mean in presence of outliers
- raw/central partial sums up to the fourth order (for brevity referred to as *raw/central sums*)
- matrix of cross-products and sums of squares (for brevity referred to as *cross-product matrix*)
- median absolute deviation, mean absolute deviation

The library also contains functions to perform the following tasks:

- Detect outliers in datasets
- Support missing values in datasets
- Parameterize correlation matrices
- Compute quantiles for streaming data

Mathematical Notation and Definitions defines the supported operations in the Summary Statistics routines.

You can access the Summary Statistics routines through the Fortran 90 and C89 language interfaces. You can use the Fortran 90 interface with programs written in Fortran 95.

The mkl_vsl.f90 header file is in the ${MKL}/include directory.

See more details about the Fortran header in Random Number Generators topic.

You can find examples that demonstrate calculation of the Summary Statistics estimates in the ${MKL}/examples/vslf example directory.

The Summary Statistics API is implemented through task objects, or tasks. A task object is a data structure, or a descriptor, holding parameters that determine a specific Summary Statistics operation. For example, such parameters may be precision, dimensions of user data, the matrix of the observations, or shapes of data arrays.

All the Summary Statistics routines process a task object as follows:

1. Create a task.
2. Modify settings of the task parameters.
3. Compute statistical estimates.
4. Destroy the task.

The Summary Statistics functions fall into the following categories:

Task Constructors - routines that create a new task object descriptor and set up most common parameters (dimension, number of observations, and matrix of the observations).

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Computation Routine - a routine that computes specified statistical estimates.

Task Destructor - a routine that deletes the task object and frees the memory.

A Summary Statistics task object contains a series of pointers to the input and output data arrays. You can read and modify the datasets and estimates at any time but you should allocate and release memory for such data.

See detailed information on the algorithms, API, and their usage in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* [SS Notes].

Summary Statistics Naming Conventions

The names of routines in the Summary Statistics are in lowercase (`vslsssseditquantiles`), while the names of types and constants are in uppercase. The names are not case-sensitive.

The names of routines have the following structure:

`vsl[datatype]ss<base name>`

where

- `vsl`is a prefix indicating that the routine belongs to Intel® oneAPI Math Kernel Library (oneMKL) Vector Statistics.
- `[datatype]` specifies the type of the input and/or output data and can be `s` (single precision real type), `d` (double precision real type), or `i` (integer type).
- `SS`/`ss` indicates that the routine is intended for calculations of the Summary Statistics estimates.
- *<base name>* specifies a particular functionality that the routine is designed for, for example, `NewTask`, `Compute`, `DeleteTask`.

NOTE
The Summary Statistics routine `vslDeleteTask` for deletion of the task is independent of the data type and its name omits the `[datatype]` field.

Summary Statistics Data Types

The Summary Statistics routines use the following data types for calculations:

Type	Data Object
Fortran 90: `TYPE(VSL_SS_TASK)`	Pointer to a Summary Statistics task
Fortran 90: `REAL(KIND=4)`	Input/output user data in single precision
Fortran 90: `REAL(KIND=8)`	Input/output user data in double precision
Fortran 90: `INTEGER` or `INTEGER(KIND=8)`	Other data

NOTE
The actual size of the generic integer type is platform-specific and can be 32 or 64 bits in length. Before you compile your application, set an appropriate size for integers. See details in the 'Using the ILP64 Interface vs. LP64 Interface' section of the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Summary Statistics Parameters

The basic parameters in the task descriptor (addresses of dimensions, number of observations, and datasets) are assigned values when the task editors create or modify the task object. Other parameters are determined by the specific task and changed by the task editors.

Summary Statistics Task Status and Error Reporting

The task status is an integer value, which is zero if no error is detected, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The header files define symbolic names for the status codes. These names are defined as integer constants via `PARAMETER` operators.

The header files define the following status codes for the Summary Statistics error codes:

Summary Statistics Status Codes

Status Code	Description
`VSL_STATUS_OK`	Operation is successfully completed.
`VSL_SS_ERROR_ALLOCATION_FAILURE`	Memory allocation has failed.
`VSL_SS_ERROR_BAD_DIMEN`	Dimension value is invalid.
`VSL_SS_ERROR_BAD_OBSERV_N`	Invalid number (zero or negative) of observations was obtained.
`VSL_SS_ERROR_STORAGE_NOT_SUPPORTED`	Storage format is not supported.
`VSL_SS_ERROR_BAD_INDC_ADDR`	Array of indices is not defined.
`VSL_SS_ERROR_BAD_WEIGHTS`	Array of weights contains negative values.
`VSL_SS_ERROR_BAD_MEAN_ADDR`	Array of means is not defined.
`VSL_SS_ERROR_BAD_2R_MOM_ADDR`	Array of the second order raw moments is not defined.
`VSL_SS_ERROR_BAD_3R_MOM_ADDR`	Array of the third order raw moments is not defined.
`VSL_SS_ERROR_BAD_4R_MOM_ADDR`	Array of the fourth order raw moments is not defined.
`VSL_SS_ERROR_BAD_2C_MOM_ADDR`	Array of the second order central moments is not defined.
`VSL_SS_ERROR_BAD_3C_MOM_ADDR`	Array of the third order central moments is not defined.
`VSL_SS_ERROR_BAD_4C_MOM_ADDR`	Array of the fourth order central moments is not defined.
`VSL_SS_ERROR_BAD_KURTOSIS_ADDR`	Array of kurtosis values is not defined.
`VSL_SS_ERROR_BAD_SKEWNESS_ADDR`	Array of skewness values is not defined.
`VSL_SS_ERROR_BAD_MIN_ADDR`	Array of minimum values is not defined.
`VSL_SS_ERROR_BAD_MAX_ADDR`	Array of maximum values is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_VARIATION_ADDR	Array of variation coefficients is not defined.
VSL_SS_ERROR_BAD_COV_ADDR	Covariance matrix is not defined.
VSL_SS_ERROR_BAD_COR_ADDR	Correlation matrix is not defined.
VSL_SS_ERROR_BAD_QUANT_ORDER_ADDR	Array of quantile orders is not defined.
VSL_SS_ERROR_BAD_QUANT_ORDER	Quantile order value is invalid.
VSL_SS_ERROR_BAD_QUANT_ADDR	Array of quantiles is not defined.
VSL_SS_ERROR_BAD_ORDER_STATS_ADDR	Array of order statistics is not defined.
VSL_SS_ERROR_MOMORDER_NOT_SUPPORTED	Moment of requested order is not supported.
VSL_SS_NOT_FULL_RANK_MATRIX	Correlation matrix is not of full rank.
VSL_SS_ERROR_ALL_OBSERVS_OUTLIERS	All observations are outliers. (At least one observation must not be an outlier.)
VSL_SS_ERROR_BAD_ROBUST_COV_ADDR	Robust covariance matrix is not defined.
VSL_SS_ERROR_BAD_ROBUST_MEAN_ADDR	Array of robust means is not defined.
VSL_SS_ERROR_METHOD_NOT_SUPPORTED	Requested method is not supported.
VSL_SS_ERROR_NULL_TASK_DESCRIPTOR	Task descriptor is null.
VSL_SS_ERROR_BAD_OBSERV_ADDR	Dataset matrix is not defined.
VSL_SS_ERROR_BAD_ACCUM_WEIGHT_ADDR	Pointer to the variable that holds the value of accumulated weight is not defined.
VSL_SS_ERROR_SINGULAR_COV	Covariance matrix is singular.
VSL_SS_ERROR_BAD_POOLED_COV_ADDR	Pooled covariance matrix is not defined.
VSL_SS_ERROR_BAD_POOLED_MEAN_ADDR	Array of pooled means is not defined.
VSL_SS_ERROR_BAD_GROUP_COV_ADDR	Group covariance matrix is not defined.
VSL_SS_ERROR_BAD_GROUP_MEAN_ADDR	Array of group means is not defined.
VSL_SS_ERROR_BAD_GROUP_INDC_ADDR	Array of group indices is not defined.
VSL_SS_ERROR_BAD_GROUP_INDC	Group indices have improper values.
VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_ADDR	Array of parameters for the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_N_ADDR	Pointer to size of the parameter array for the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_OUTLIERS_WEIGHTS_ADDR	Output of the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_ADDR	Array of parameters of the robust covariance estimation algorithm is not defined.
VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_N_ADDR	Pointer to the number of parameters of the algorithm for robust covariance is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_STORAGE_ADDR	Pointer to the variable that holds the storage format is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COV_IDX_ADDR	Array that encodes sub-components of a random vector for the partial covariance algorithm is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COV_IDX	Array that encodes sub-components of a random vector for partial covariance has improper values.
VSL_SS_ERROR_BAD_PARTIAL_COV_ADDR	Partial covariance matrix is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COR_ADDR	Partial correlation matrix is not defined.
VSL_SS_ERROR_BAD_MI_PARAMS_ADDR	Array of parameters for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_PARAMS_N_ADDR	Pointer to number of parameters for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_BAD_PARAMS_N	Size of the parameter array of the Multiple Imputation method is invalid.
VSL_SS_ERROR_BAD_MI_PARAMS	Parameters of the Multiple Imputation method are invalid.
VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_N_ADDR	Pointer to the number of initial estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_ADDR	Array of initial estimates for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_ADDR	Array of simulated missing values in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N_ADDR	Pointer to the size of the array of simulated missing values in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_ESTIMATES_N_ADDR	Pointer to the number of parameter estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_ESTIMATES_ADDR	Array of parameter estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N	Invalid size of the array of simulated values in the Multiple Imputation method.
VSL_SS_ERROR_BAD_MI_ESTIMATES_N	Invalid size of an array to hold parameter estimates obtained using the Multiple Imputation method.
VSL_SS_ERROR_BAD_MI_OUTPUT_PARAMS	Array of output parameters in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_PRIOR_N_ADDR	Pointer to the number of prior parameters is not defined.
VSL_SS_ERROR_BAD_MI_PRIOR_ADDR	Array of prior parameters is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_MI_MISSING_VALS_N	Invalid number of missing values was obtained.
VSL_SS_SEMIDEFINITE_COR	Correlation matrix passed into the parameterization function is semi-definite.
VSL_SS_ERROR_BAD_PARAMTR_COR_ADDR	Correlation matrix to be parameterized is not defined.
VSL_SS_ERROR_BAD_COR	All eigenvalues of the correlation matrix to be parameterized are non-positive.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N_ADDR	Pointer to the number of parameters for the quantile computation algorithm for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_ADDR	Array of parameters of the quantile computation algorithm for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N	Invalid number of parameters of the quantile computation algorithm for streaming data has been obtained.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS	Invalid parameters of the quantile computation algorithm for streaming data have been passed.
VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER_ADDR	Array of the quantile orders for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER	Invalid quantile order for streaming data is defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_ADDR	Array of quantiles for streaming data is not defined.
VSL_SS_ERROR_BAD_SUM_ADDR	Array of sums is not defined.
VSL_SS_ERROR_BAD_2R_SUM_ADDR	Array of raw sums of 2nd order is not defined.
VSL_SS_ERROR_BAD_3R_SUM_ADDR	Array of raw sums of 3rd order is not defined.
VSL_SS_ERROR_BAD_4R_SUM_ADDR	Array of raw sums of 4th order is not defined.
VSL_SS_ERROR_BAD_2C_SUM_ADDR	Array of central sums of 2nd order is not defined.
VSL_SS_ERROR_BAD_3C_SUM_ADDR	Array of central sums of 3rd order is not defined.
VSL_SS_ERROR_BAD_4C_SUM_ADDR	Array of central sums of 4th order is not defined.
VSL_SS_ERROR_BAD_CP_SUM_ADDR	Cross-product matrix is not defined.
VSL_SS_ERROR_BAD_MDAD_ADDR	Array of median absolute deviations is not defined.
VSL_SS_ERROR_BAD_MNAD_ADDR	Array of mean absolute deviations is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_SORTED_OBSERV_ADDR	Array for storing observation sorting results is not defined.
VSL_SS_ERROR_ERROR_INDICES_NOT_SUPPORTED	Array of indices is not supported.

Routines for robust covariance estimation, outlier detection, partial covariance estimation, multiple imputation, and parameterization of a correlation matrix can return internal error codes that are related to a specific implementation. Such error codes indicate invalid input data or other bugs in the Intel® oneAPI Math Kernel Library (oneMKL) routines other than the Summary Statistics routines.

Summary Statistics Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters.

NOTE
If the constructor fails to create a task descriptor, it returns the NULL task pointer.

vslSSNewTask

Creates and initializes a new summary statistics task descriptor.

Syntax

status = vslsssnewtask(*task*, *p*, *n*, *xstorage*, *x*, *w*, *indices*)

status = vsldssnewtask(*task*, *p*, *n*, *xstorage*, *x*, *w*, *indices*)

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
p	INTEGER	Dimension of the task, number of variables
n	INTEGER	Number of observations
xstorage	INTEGER	Storage format of matrix of observations
x	REAL(KIND=4) DIMENSION(*) for vslsssnewtask REAL(KIND=8) DIMENSION(*) for vsldssnewtask	Matrix of observations
w	REAL(KIND=4) DIMENSION(*) for vslsssnewtask REAL(KIND=8) DIMENSION(*) for vsldssnewtask	Array of weights of size *n*. Elements of the arrays are non-negative numbers. If a NULL pointer is passed, each observation is assigned weight equal to 1.

Name	Type	Description
indices	INTEGER, DIMENSION(*)	Array of vector components that will be processed. Size of array is p. If a NULL pointer is passed, all components of random vector are processed.

Output Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
status	INTEGER	Set to VSL_STATUS_OK if the task is created successfully, otherwise a non-zero error code is returned.

Description

Each vslSSNewTask constructor routine creates a new summary statistics task descriptor with the user-specified value for a required parameter, dimension of the task. The optional parameters (matrix of observations, its storage format, number of observations, weights of observations, and indices of the random vector components) are set to their default values.

The observations of random p-dimensional vector $\xi = (\xi_1, ..., \xi_i, ..., \xi_p)$, which are n vectors of dimension p, are passed as a one-dimensional array x. The parameter *xstorage* defines the storage format of the observations and takes one of the possible values listed in Table "Storage format of matrix of observations and order statistics".

Storage format of matrix of observations, order statistics, and matrix of sorted observations

Parameter	Description
VSL_SS_MATRIX_STORAGE_ROWS	The observations of random vector ξ are packed by rows: n data points for the vector component ξ_1 come first, n data points for the vector component ξ_2 come second, and so forth.
VSL_SS_MATRIX_STORAGE_COLS	The observations of random vector ξ are packed by columns: the first p-dimensional observation of the vector ξ comes first, the second p-dimensional observation of the vector comes second, and so forth.

NOTE
Since matrices in Fortran are stored by columns while in C they are stored by rows, initialization of the *xstorage* variable in Fortran is opposite to that in C. Set *xstorage* to VSL_SS_MATRIX_STORAGE_COLS, if the dataset is stored as a two-dimensional matrix that consists of p rows and n columns; otherwise, use the VSL_SS_MATRIX_STORAGE_ROWS constant.

A one-dimensional array w of size n contains non-negative weights assigned to the observations. You can pass a NULL array into the constructor. In this case, each observation is assigned the default value of the weight.

You can choose vector components for which you wish to compute statistical estimates. If an element of the vector indices of size p contains 0, the observations that correspond to this component are excluded from the calculations. If you pass the `NULL` value of the parameter into the constructor, statistical estimates for all random variables are computed.

If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

Summary Statistics Task Editors

Task editors are intended to set up or change the task parameters listed in Table "Parameters of Summary Statistics Task to Be Initialized or Modified". As an example, to compute the sample mean for a one-dimensional dataset, initialize a variable for the mean value, and pass its address into the task as shown in the example below:

```
#define DIM    1
#define N   1000

int main()
{
     VSLSSTaskPtr task;
     double x[N];
     double mean;
     MKL_INT p, n, xstorage;
     int status;
     /* initialize variables used in the computations of sample mean */
     p = DIM;
     n = N;
     xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
     mean = 0.0;

     /* create task */
     status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

     /* initialize task parameters */
     status = vsldSSEditTask( task, VSL_SS_ED_MEAN, &mean );

     /* compute mean using SS fast method */
     status = vsldSSCompute(task, VSL_SS_MEAN, VSL_SS_METHOD_FAST );

     /* deallocate task resources */
     status = vslSSDeleteTask( &task );

     return 0;
}
```

Use the single (`vslsssedittask`) or double (`vsldsseditask`) version of an editor, to initialize single or double precision version task parameters, respectively. Use an integer version of an editor (`vslisseditask`) to initialize parameters of the integer type.

Table "Summary Statistics Task Editors" lists the task editors for Summary Statistics. Each of them initializes and/or modifies a respective group of related parameters.

Summary Statistics Task Editors

Editor	Description
`vslSSEditTask`	Changes a pointer in the task descriptor.
`vslSSEditMoments`	Changes pointers to arrays associated with raw and central moments.

Editor	Description
vslSSEditSums	Modifies the pointers to arrays that hold sum estimates.
vslSSEditCovCor	Changes pointers to arrays associated with covariance and/or correlation matrices.
vslSSEditCP	Modifies the pointers to cross-product matrix parameters.
vslSSEditPartialCovCor	Changes pointers to arrays associated with partial covariance and/or correlation matrices.
vslSSEditQuantiles	Changes pointers to arrays associated with quantile/order statistics calculations.
vslSSEditStreamQuantiles	Changes pointers to arrays for quantile related calculations for streaming data.
vslSSEditPooledCovariance	Changes pointers to arrays associated with algorithms related to a pooled covariance matrix.
vslSSEditRobustCovariance	Changes pointers to arrays for robust estimation of a covariance matrix and mean.
vslSSEditOutliersDetection	Changes pointers to arrays for detection of outliers.
vslSSEditMissingValues	Changes pointers to arrays associated with the method of supporting missing values in a dataset.
vslSSEditCorParameterization	Changes pointers to arrays associated with the algorithm for parameterization of a correlation matrix.

NOTE
You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vslSSEditTask

Modifies address of an input/output parameter in the task descriptor.

Syntax

```
status = vslsssedittask(task, parameter, par_addr)
status = vsldssedittask(task, parameter, par_addr)
status = vslissedittask(task, parameter, par_addr)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
task	`TYPE(VSL_SS_TASK)`	Descriptor of the task
parameter	`INTEGER`	Parameter to change

Name	Type	Description

par_addr	`REAL(KIND=4) DIMENSION(*)` for `vslsssedittask`
	`REAL(KIND=8) DIMENSION(*)` for `vsldssedittask`
	`INTEGER DIMENSION(*)` for `vslissedittask`
		Address of the new parameter

Output Parameters

Name	Type	Description
status	`INTEGER`	Current status of the task

Description

The `vslSSEditTask` routine replaces the pointer to the parameter stored in the Summary Statistics task descriptor with the *par_addr* pointer. If you pass the `NULL` pointer to the editor, no changes take place in the task and a corresponding error code is returned. See Table "Parameters of Summary Statistics Task to Be Initialized or Modified" for the predefined values of the parameter.

Use the single (`vslsssedittask`) or double (`vsldssedittask`) version of the editor, to initialize single or double precision version task parameters, respectively. Use an integer version of the editor (`vslissedittask`) to initialize parameters of the integer type.

Parameters of Summary Statistics Task to Be Initialized or Modified

Parameter Value	Type	Purpose	Initialization
`VSL_SS_ED_DIMEN`	`i`	Address of a variable that holds the task dimension	Required. Positive integer value.
`VSL_SS_ED_OBSERV_N`	`i`	Address of a variable that holds the number of observations	Required. Positive integer value.
`VSL_SS_ED_OBSERV`	`d`, `s`	Address of the observation matrix	Required. Provide the matrix containing your observations.
`VSL_SS_ED_OBSERV_STORAGE`	`i`	Address of a variable that holds the storage format for the observation matrix	Required. Provide a storage format supported by the library whenever you pass a matrix of observations.[1]
`VSL_SS_ED_INDC`	`i`	Address of the array of indices	Optional. Provide this array if you need to process individual components of the random vector. Set entry *i* of the array to one to include the *i*th coordinate in the analysis. Set entry *i* of the array to zero to exclude the *i*th coordinate from the analysis.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_WEIGHTS	d, s	Address of the array of observation weights	Optional. If the observations have weights different from the default weight (one), set entries of the array to non-negative floating point values.
VSL_SS_ED_MEAN	d, s	Address of the array of means	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2R_MOM	d, s	Address of an array of raw moments of the second order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3R_MOM	d, s	Address of an array of raw moments of the third order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4R_MOM	d, s	Address of an array of raw moments of the fourth order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2C_MOM	d, s	Address of an array of central moments of the second order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.
VSL_SS_ED_3C_MOM	d, s	Address of an array of central moments of the third order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.
VSL_SS_ED_4C_MOM	d, s	Address of an array of central moments of the fourth order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make

Parameter Value	Type	Purpose	Initialization
			sure you also provide arrays for raw moments of the first, second, third, and fourth order.
VSL_SS_ED_KURTOSIS	d, s	Address of the array of kurtosis estimates	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, third, and fourth order.
VSL_SS_ED_SKEWNESS	d, s	Address of the array of skewness estimates	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.
VSL_SS_ED_MIN	d, s	Address of the array of minimum estimates	Optional. Set entries of array to meaningful values, such as the values of the first observation.
VSL_SS_ED_MAX	d, s	Address of the array of maximum estimates	Optional. Set entries of array to meaningful values, such as the values of the first observation.
VSL_SS_ED_VARIATION	d, s	Address of the array of variation coefficients	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.
VSL_SS_ED_COV	d, s	Address of a covariance matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Make sure you also provide an array for the mean.
VSL_SS_ED_COV_STORAGE	i	Address of the variable that holds the storage format for a covariance matrix	Required. Provide a storage format supported by the library whenever you intend to compute the covariance matrix.[2]
VSL_SS_ED_COR	d, s	Address of a correlation matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a

Parameter Value	Type	Purpose	Initialization
			progressive estimate. If you initialize the matrix in non-trivial way, make sure that the main diagonal contains variance values. Also, provide an array for the mean.
VSL_SS_ED_COR_STORAGE	i	Address of the variable that holds the correlation storage format for a correlation matrix	Required. Provide a storage format supported by the library whenever you intend to compute the correlation matrix.[2]
VSL_SS_ED_ACCUM_WEIGHT	d, s	Address of the array of size 2 that holds the accumulated weight (sum of weights) in the first position and the sum of weights squared in the second position	Optional. Set the entries of the matrix to meaningful values (typically zero) if you intend to do progressive processing of the dataset or need the sum of weights and sum of squared weights assigned to observations.
VSL_SS_ED_QUANT_ORDER_N	i	Address of the variable that holds the number of quantile orders	Required. Positive integer value. Provide the number of quantile orders whenever you compute quantiles.
VSL_SS_ED_QUANT_ORDER	d, s	Address of the array of quantile orders	Required. Set entries of array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.
VSL_SS_ED_QUANT_QUANTILES	d, s	Address of the array of quantiles	None.
VSL_SS_ED_ORDER_STATS	d, s	Address of the array of order statistics	None.
VSL_SS_ED_GROUP_INDC	i	Address of the array of group indices used in computation of a pooled covariance matrix	Required. Set entry *i* to integer value *k* if the observation belongs to group *k*. Values of *k* take values in the range [0, *g*-1], where *g* is the number of groups.
VSL_SS_ED_POOLED_COV_STORAGE	i	Address of a variable that holds the storage format for a pooled covariance matrix	Required. Provide a storage format supported by the library whenever you intend to compute pooled covariance.[2]
VSL_SS_ED_POOLED_MEAN	d, s	Address of an array of pooled means	None.
VSL_SS_ED_POOLED_COV	d, s	Address of pooled covariance matrices	None.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_GROUP_COV_INDC	i	Address of an array of indices for which covariance/means should be computed	Optional. Set the *k*th entry of the array to 1 if you need group covariance and mean for group *k*; otherwise set it to zero.
VSL_SS_ED_REQ_GROUP_INDC	i	Address of an array of indices for which group estimates such as covariance or means are requested	Optional. Set the *k*th entry of the array to 1 if you need an estimate for group *k*; otherwise set it to zero.
VSL_SS_ED_GROUP_MEANS	i	Address of an array of group means	None.
VSL_SS_ED_GROUP_COV_STORAGE	d, s	Address of a variable that holds the storage format for a group covariance matrix	Required. Provide a storage format supported by the library whenever you intend to get group covariance.[2]
VSL_SS_ED_GROUP_COV	d, s	Address of group covariance matrices	None.
VSL_SS_ED_ROBUST_COV_STORAGE	d, s	Address of a variable that holds the storage format for a robust covariance matrix	Required. Provide a storage format supported by the library whenever you compute robust covariance[2].
VSL_SS_ED_ROBUST_COV_PARAMS_N	i	Address of a variable that holds the number of algorithmic parameters of the method for robust covariance estimation	Required. Set to the number of TBS parameters, *VSL_SS_TBS_PARAMS_N*.
VSL_SS_ED_ROBUST_COV_PARAMS	d, s	Address of an array of parameters of the method for robust estimation of a covariance	Required. Set the entries of the array according to the description in vslSSEditRobustCovariance.
VSL_SS_ED_ROBUST_MEAN	i	Address of an array of robust means	None.
VSL_SS_ED_ROBUST_COV	d, s	Address of a robust covariance matrix	None.
VSL_SS_ED_OUTLIERS_PARAMS_N	d, s	Address of a variable that holds the number of parameters of the outlier detection method	Required. Set to the number of outlier detection parameters, *VSL_SS_BACON_PARAMS_N*.
VSL_SS_ED_OUTLIERS_PARAMS	i	Address of an array of algorithmic parameters for the outlier detection method	Required. Set the entries of the array according to the description in vslSSEditOutliersDetection.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_OUTLIERS_WEIGHT	d, s	Address of an array of weights assigned to observations by the outlier detection method	None.
VSL_SS_ED_ORDER_STATS_STORAGE	d, s	Address of a variable that holds the storage format of an order statistics matrix	Required. Provide a storage format supported by the library whenever you compute a matrix of order statistics.[1]
VSL_SS_ED_PARTIAL_COV_IDX	i	Address of an array that encodes subcomponents of a random vector	Required. Set the entries of the array according to the description in vslSSEditPartialCovCor.
VSL_SS_ED_PARTIAL_COV	d, s	Address of a partial covariance matrix	None.
VSL_SS_ED_PARTIAL_COV_STORAGE	i	Address of a variable that holds the storage format of a partial covariance matrix	Required. Provide a storage format supported by the library whenever you compute the partial covariance.[2]
VSL_SS_ED_PARTIAL_COR	d, s	Address of a partial correlation matrix	None.
VSL_SS_ED_PARTIAL_COR_STORAGE	i	Address of a variable that holds the storage format for a partial correlation matrix	Required. Provide a storage format supported by the library whenever you compute the partial correlation.[2]
VSL_SS_ED_MI_PARAMS_N	i	Address of a variable that holds the number of algorithmic parameters for the Multiple Imputation method	Required. Set to the number of MI parameters, *VSL_SS_MI_PARAMS_SIZE*.
VSL_SS_ED_MI_PARAMS	d, s	Address of an array of algorithmic parameters for the Multiple Imputation method	Required. Set entries of the array according to the description in vslSSEditMissingValues.
VSL_SS_ED_MI_INIT_ESTIMATES_N	i	Address of a variable that holds the number of initial estimates for the Multiple Imputation method	Optional. Set to $p+p*(p+1)/2$, where p is the task dimension.
VSL_SS_ED_MI_INIT_ESTIMATES	d, s	Address of an array of initial estimates for the Multiple Imputation method	Optional. Set the values of the array according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics" in the *Intel® oneAPI Math Kernel Library*

Parameter Value	Type	Purpose	Initialization
			(oneMKL) Summary Statistics Application Notes document [SS Notes].
VSL_SS_ED_MI_SIMUL_VALS_N	i	Address of a variable that holds the number of simulated values in the Multiple Imputation method	Optional. Positive integer indicating the number of missing points in the observation matrix.
VSL_SS_ED_MI_SIMUL_VALS	d, s	Address of an array of simulated values in the Multiple Imputation method	None.
VSL_SS_ED_MI_ESTIMATES_N	i	Address of a variable that holds the number of estimates obtained as a result of the Multiple Imputation method	Optional. Positive integer number defined according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics" in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [SS Notes].
VSL_SS_ED_MI_ESTIMATES	d, s	Address of an array of estimates obtained as a result of the Multiple Imputation method	None.
VSL_SS_ED_MI_PRIOR_N	i	Address of a variable that holds the number of prior parameters for the Multiple Imputation method	Optional. If you pass a user-defined array of prior parameters, set this parameter to $(p^2+3*p+4)/2$, where p is the task dimension.
VSL_SS_ED_MI_PRIOR	d, s	Address of an array of prior parameters for the Multiple Imputation method	Optional. Set entries of the array of prior parameters according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics" in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [SS Notes].
VSL_SS_ED_PARAMTR_COR	d, s	Address of a parameterized correlation matrix	None.
VSL_SS_ED_PARAMTR_COR_STORAGE	i	Address of a variable that holds the storage format of a parameterized correlation matrix	Required. Provide a storage format supported by the library whenever you compute the parameterized correlation matrix.[2]

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_STREAM_QUANT_PARAMS_N	i	Address of a variable that holds the number of parameters of a quantile computation method for streaming data	Required. Set to the number of quantile computation parameters, *VSL_SS_SQUANTS_ZW_PARAMS_N*.
VSL_SS_ED_STREAM_QUANT_PARAMS	d, s	Address of an array of parameters of a quantile computation method for streaming data	Required. Set the entries of the array according to the description in "Computing Quantiles for Streaming Data" in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [SS Notes].
VSL_SS_ED_STREAM_QUANT_ORDER_N	i	Address of a variable that holds the number of quantile orders for streaming data	Required. Positive integer value.
VSL_SS_ED_STREAM_QUANT_ORDER	d, s	Address of an array of quantile orders for streaming data	Required. Set entries of the array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.
VSL_SS_ED_STREAM_QUANT_QUANTILES	d, s	Address of an array of quantiles for streaming data	None.
VSL_SS_ED_SUM	d, s	Address of array of sums	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2R_SUM	d, s	Address of array of raw sums of 2nd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3R_SUM	d, s	Address of array of raw sums of 3rd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4R_SUM	d, s	Address of array of raw sums of 4th order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_2C_SUM	d, s	Address of array of central sums of 2nd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3C_SUM	d, s	Address of array of central sums of 3rd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4C_SUM	d, s	Address of array of central sums of 4th order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_CP	d, s	Address of cross-product matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_MDAD	d, s	Address of array of median absolute deviations	None.
VSL_SS_ED_MNAD	d, s	Address of array of mean absolute deviations	None.
VSL_SS_ED_SORTED_OBSERV	d, s	Address of the array that stores sorted results	None.
VSL_SS_ED_SORTED_OBSERV_STORAGE	i	Address of a variable that holds the storage format of an output matrix	Required. Provide a supported storage format whenever you specify sorting of the observation matrix.

1. See Table: "Storage format of matrix of observations and order statistics" for storage formats.
2. See Table: "Storage formats of a variance-covariance/correlation matrix" for storage formats.

vslSSEditMoments

Modifies the pointers to arrays that hold moment estimates.

Syntax

```
status = vslssseditmoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m)
status = vsldsseditmoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m)
```

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
mean	REAL(KIND=4) DIMENSION(*) for vslssseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of means
r2m	REAL(KIND=4) DIMENSION(*) for vslssseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of raw moments of the 2nd order
r3m	REAL(KIND=4) DIMENSION(*) for vslssseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of raw moments of the 3rd order
r4m	REAL(KIND=4) DIMENSION(*) for vslssseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of raw moments of the 4th order
c2m	REAL(KIND=4) DIMENSION(*) for vslssseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of central moments of the 2nd order
c3m	REAL(KIND=4) DIMENSION(*) for vslssseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of central moments of the 3rd order
c4m	REAL(KIND=4) DIMENSION(*) for vslssseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of central moments of the 4th order

Output Parameters

Name	Type	Description
status	INTEGER	Current status of the task

Description

The vslSSEditMoments routine replaces pointers to the arrays that hold estimates of raw and central moments with values passed as corresponding parameters of the routine. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vslSSEditSums

Modifies the pointers to arrays that hold sum estimates.

Syntax

```
status = vslssseditsums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s)
status = vsldsseditsums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s)
```

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
sum	REAL(KIND=4) DIMENSION(*) for vslssseditsums REAL(KIND=8) DIMENSION(*) for vsldsseditsums	Pointer to the array of sums
r2s	REAL(KIND=4) DIMENSION(*) for vslssseditsums REAL(KIND=8) DIMENSION(*) for vsldsseditsums	Pointer to the array of raw sums of the second order
r3s	REAL(KIND=4) DIMENSION(*) for vslssseditsums REAL(KIND=8) DIMENSION(*) for vsldsseditsums	Pointer to the array of raw sums of the third order
r4s	REAL(KIND=4) DIMENSION(*) for vslssseditsums REAL(KIND=8) DIMENSION(*) for vsldsseditsums	Pointer to the array of raw sums of the fourth order
c2s	REAL(KIND=4) DIMENSION(*) for vslssseditsums REAL(KIND=8) DIMENSION(*) for vsldsseditsums	Pointer to the array of central sums of the second order
c3s	REAL(KIND=4) DIMENSION(*) for vslssseditsums REAL(KIND=8) DIMENSION(*) for vsldsseditsums	Pointer to the array of central sums of the third order
c4s	REAL(KIND=4) DIMENSION(*) for vslssseditsums REAL(KIND=8) DIMENSION(*) for vsldsseditsums	Pointer to the array of central sums of the fourth order

Output Parameters

Name	Type	Description
status	INTEGER	Current status of the task

Description

The vslSSEditSums routine replaces pointers to the arrays that hold estimates of raw and central sums with values passed as corresponding parameters of the routine. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vslSSEditCovCor

Modifies the pointers to covariance/correlation/cross-product parameters.

Syntax

status = vslssseditcovcor(*task*, *mean*, *cov*, *cov_storage*, *cor*, *cor_storage*)

status = vsldsseditcovcor(*task*, *mean*, *cov*, *cov_storage*, *cor*, *cor_storage*)

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
mean	REAL(KIND=4) DIMENSION(*) for vslssseditcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditcovcor	Pointer to the array of means
cov	REAL(KIND=4) DIMENSION(*) for vslssseditcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditcovcor	Pointer to a covariance matrix
cov_storage	INTEGER	Pointer to the storage format of the covariance matrix
cor	REAL(KIND=4) DIMENSION(*) for vslssseditcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditcovcor	Pointer to a correlation matrix
cor_storage	INTEGER	Pointer to the storage format of the correlation matrix

Output Parameters

Name	Type	Description
status	INTEGER	Current status of the task

Description

The `vslSSEditCovCor` routine replaces pointers to the array of means, covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

The storage parameters, *cov_storage* and *cor_storage*, describe the storage format used for the *p*-by-*p* symmetric variance-covariance/correlation/cross-product matrix *C*. The matrix *C* can be described as

$$C = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & \cdots & \cdots & c_{1,p} \\ c_{2,1} & c_{2,2} & \cdots & \cdots & \cdots & c_{2,p} \\ \vdots & \vdots & \ddots & & & \vdots \\ \vdots & \vdots & & c_{i,j} & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ c_{p,1} & c_{p,2} & \cdots & \cdots & \cdots & c_{p,p} \end{pmatrix}$$

Table "Storage formats of a variance-covariance/correlation/cross-product matrix" shows how the matrix is stored in a one-dimensional array *cp* for different values of the storage parameters.

Storage formats of variance-covariance/correlation/cross-product matrices

Parameter	Description
`VSL_SS_MATRIX_STORAGE_FULL`	The array *cp* contains all elements of the matrix stored sequentially, column-by-column: *cp*(1) contains $c_{1,1}$ *cp*(2) contains $c_{2,1}$ *cp*(*p*) contains $c_{p,1}$ *cp*(*p* + 1) contains $c_{1,2}$ *cp*(*p***p*) contains $c_{p,p}$ The size of array *cp* is *p***p*.
`VSL_SS_MATRIX_STORAGE_L_PACKED`	The array *cp* contains the lower triangular part of the symmetric matrix stored sequentially, column-by-column: *cp*(1) contains $c_{1,1}$ *cp*(2) contains $c_{2,1}$ *cp*(3) contains $c_{3,1}$ and so on. The size of the array is *p**(*p*+ 1)/2.
`VSL_SS_MATRIX_STORAGE_U_PACKED`	The array *cp* contains the upper triangular part of the symmetric matrix stored sequentially, column-by-column: *cp*(1) contains $c_{1,1}$ *cp*(2) contains $c_{1,2}$

Parameter	Description
	`cp(3)` contains $c_{2,2}$ and so on. The size of the array is $p*(p+1)/2$.

vslSSEditCP

Modifies the pointers to cross-product matrix parameters.

Syntax

```
status = vslssseditcp(task, mean, sum, cp, cp_storage)
status = vsldsseditcp(task, mean, sum, cp, cp_storage)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
task	`TYPE(VSL_SS_TASK)`	Descriptor of the task
mean	`REAL(KIND=4) DIMENSION(*)` for `vslssseditcp` `REAL(KIND=8) DIMENSION(*)` for `vsldsseditcp`	Pointer to array of means
sum	`REAL(KIND=4) DIMENSION(*)` for `vslssseditcp` `REAL(KIND=8) DIMENSION(*)` for `vsldsseditcp`	Pointer to array of sums
cp	`REAL(KIND=4) DIMENSION(*)` for `vslssseditcp` `REAL(KIND=8) DIMENSION(*)` for `vsldsseditcp`	Pointer to a cross-product matrix
cp_storage	`INTEGER`	Pointer to the storage format of the cross-product matrix

Output Parameters

Name	Type	Description
status	`INTEGER`	Current status of the task

Description

The `vslSSEditCP` routine replaces pointers to the array of means, array of sums, cross-product matrix, and its storage format with values passed as corresponding parameters of the routine. See Table: "Storage formats of a variance-covariance/correlation/cross-product matrix" for possible values of the *cp_storage* parameter. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Storage formats of variance-covariance/correlation/cross-product matrices

Parameter	Description
`VSL_SS_MATRIX_STORAGE_FULL`	The array *cp* contains all elements of the matrix stored sequentially, column-by-column: *cp*(1) contains $c_{1,1}$ *cp*(2) contains $c_{2,1}$ *cp*(*p*) contains $c_{p,1}$ *cp*(*p* + 1) contains $c_{1,2}$ *cp*(*p***p*) contains $c_{p,p}$ The size of array *cp* is *p***p*.
`VSL_SS_MATRIX_STORAGE_L_PACKED`	The array *cp* contains the lower triangular part of the symmetric matrix stored sequentially, column-by-column: *cp*(1) contains $c_{1,1}$ *cp*(2) contains $c_{2,1}$ *cp*(3) contains $c_{3,1}$ and so on. The size of the array is *p**(*p*+ 1)/2.
`VSL_SS_MATRIX_STORAGE_U_PACKED`	The array *cp* contains the upper triangular part of the symmetric matrix stored sequentially, column-by-column: *cp*(1) contains $c_{1,1}$ *cp*(2) contains $c_{1,2}$ *cp*(3) contains $c_{2,2}$ and so on. The size of the array is *p**(*p*+ 1)/2.

vslSSEditPartialCovCor

Modifies the pointers to partial covariance/correlation parameters.

Syntax

status = vslsssedit partialcovcor(*task*, *p_idx_array*, *cov*, *cov_storage*, *cor*, *cor_storage*, *p_cov*, *p_cov_storage*, *p_cor*, *p_cor_storage*)

status = vsldsseditpartialcovcor(*task*, *p_idx_array*, *cov*, *cov_storage*, *cor*, *cor_storage*, *p_cov*, *p_cov_storage*, *p_cor*, *p_cor_storage*)

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
p_idx_array	INTEGER	Pointer to the array that encodes indices of subcomponents *Z* and *Y* of the random vector as described in section Mathematical Notation and Definitions. *p_idx_array[i]* equals to -1 if the *i*-th component of the random vector belongs to *Z* 1, if the *i*-th component of the random vector belongs to *Y*.
cov	REAL(KIND=4) DIMENSION(*) for vslsssseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor	Pointer to a covariance matrix
cov_storage	INTEGER	Pointer to the storage format of the covariance matrix
cor	REAL(KIND=4) DIMENSION(*) for vslsssseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor	Pointer to a correlation matrix
cor_storage	INTEGER	Pointer to the storage format of the correlation matrix
p_cov	REAL(KIND=4) DIMENSION(*) for vslsssseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor	Pointer to a partial covariance matrix
p_cov_storage	INTEGER	Pointer to the storage format of the partial covariance matrix
p_cor	REAL(KIND=4) DIMENSION(*) for vslsssseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor	Pointer to a partial correlation matrix
p_cor_storage	INTEGER	Pointer to the storage format of the partial correlation matrix

Output Parameters

Name	Type	Description
status	INTEGER	Current status of the task

Description

The vslSSEditPartialCovCor routine replaces pointers to covariance/correlation arrays, partial covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. See Table "Storage formats of a variance-covariance/correlation matrix" for possible values of the *cov_storage*, *cor_storage*, *p_cov_storage*, and *p_cor_storage* parameters. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vslSSEditQuantiles

Modifies the pointers to parameters related to quantile computations.

Syntax

status = vslssseditquantiles(*task*, *quant_order_n*, *quant_order*, *quants*, *order_stats*, *order_stats_storage*)

status = vsldsseditquantiles(*task*, *quant_order_n*, *quant_order*, *quants*, *order_stats*, *order_stats_storage*)

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
quant_order_n	INTEGER	Pointer to the number of quantile orders
quant_order	REAL(KIND=4) DIMENSION(*) for vslssseditquantiles REAL(KIND=8) DIMENSION(*) for vsldsseditquantiles	Pointer to the array of quantile orders
quants	REAL(KIND=4) DIMENSION(*) for vslssseditquantiles REAL(KIND=8) DIMENSION(*) for vsldsseditquantiles	Pointer to the array of quantiles
order_stats	REAL(KIND=4) DIMENSION(*) for vslssseditquantiles REAL(KIND=8) DIMENSION(*) for vsldsseditquantiles	Pointer to the array of order statistics
order_stats_storage	INTEGER	Pointer to the storage format of the order statistics array

Output Parameters

Name	Type	Description
status	INTEGER	Current status of the task

Description

The vslSSEditQuantiles routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the array that holds order statistics, and the storage format for the order statistics with values passed into the routine. See Table "Storage format of matrix of observations and order statistics" for possible values of the *order_statistics_storage* parameter. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vslSSEditStreamQuantiles

Modifies the pointers to parameters related to quantile computations for streaming data.

Syntax

status = vslsssseditstreamquantiles(*task*, *quant_order_n*, *quant_order*, *quants*, *nparams*, *params*)

status = vsldsseditstreamquantiles(*task*, *quant_order_n*, *quant_order*, *quants*, *nparams*, *params*)

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
quant_order_n	INTEGER	Pointer to the number of quantile orders
quant_order	REAL(KIND=4) DIMENSION(*) for vslsssseditstreamquantiles REAL(KIND=8) DIMENSION(*) for vsldsseditstreamquantiles	Pointer to the array of quantile orders
quants	REAL(KIND=4) DIMENSION(*) for vslsssseditstreamquantiles REAL(KIND=8) DIMENSION(*) for vsldsseditstreamquantiles	Pointer to the array of quantiles
nparams	INTEGER	Pointer to the number of the algorithm parameters
params	REAL(KIND=4) DIMENSION(*) for vslsssseditstreamquantiles REAL(KIND=8) DIMENSION(*)	Pointer to the array of the algorithm parameters

Name	Type	Description
	for `vsldsseditstreamquantiles`	

Output Parameters

Name	Type	Description
status	`INTEGER`	Current status of the task

Description

The `vslSSEditStreamQuantiles` routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the number of the algorithm parameters, and the array of the algorithm parameters with values passed into the routine. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vslSSEditPooledCovariance

Modifies pooled/group covariance matrix array pointers.

Syntax

status = vslssseditpooledcovariance(*task*, *grp_indices*, *pld_mean*, *pld_cov*, *req_grp_indices*, *grp_means*, *grp_cov*)

status = vsldsseditpooledcovariance(*task*, *grp_indices*, *pld_mean*, *pld_cov*, *req_grp_indices*, *grp_means*, *grp_cov*)

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
task	`TYPE(VSL_SS_TASK)`	Descriptor of the task
grp_indices	`INTEGER DIMENSION(*)`	Pointer to an array of size *n*. The *i*-th element of the array contains the number of the group the observation belongs to.
pld_mean	`REAL(KIND=4) DIMENSION(*)` for `vslssseditpooledcovariance` `REAL(KIND=8) DIMENSION(*)` for `vsldsseditpooledcovariance`	Pointer to the array of pooled means
pld_cov	`REAL(KIND=4) DIMENSION(*)` for `vslssseditpooledcovariance` `REAL(KIND=8) DIMENSION(*)` for `vsldsseditpooledcovariance`	Pointer to the array that holds a pooled covariance matrix
req_grp_indices	`INTEGER DIMENSION(*)`	Pointer to the array that contains indices of groups for which estimates to return (such as covariance and mean)

Name	Type	Description
grp_means	REAL(KIND=4) DIMENSION(*) for product=Fortran vslssseditpooledcovariance REAL(KIND=8) DIMENSION(*) for vsldsseditpooledcovariance	Pointer to the array of group means
grp_cov	REAL(KIND=4) DIMENSION(*) for vslssseditpooledcovariance REAL(KIND=8) DIMENSION(*) for vsldsseditpooledcovariance	Pointer to the array that holds group covariance matrices

Output Parameters

Name	Type	Description
status	INTEGER	Current status of the task

Description

The vslSSEditPooledCovariance routine replaces pointers to the array of group indices, the array of pooled means, the array for a pooled covariance matrix, and pointers to the array of indices of group matrices, the array of group means, and the array for group covariance matrices with values passed in the editors. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.. Use the vslSSEditTask routine to replace the storage format for pooled and group covariance matrices.

vslSSEditRobustCovariance

Modifies pointers to arrays related to a robust covariance matrix.

Syntax

```
status = vslssseditrobustcovariance(task, rcov_storage, nparams, params, rmean, rcov)
status = vsldsseditrobustcovariance(task, rcov_storage, nparams, params, rmean, rcov)
```

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
rcov_storage	INTEGER	Pointer to the storage format of a robust covariance matrix
nparams	INTEGER	Pointer to the number of method parameters
params	REAL(KIND=4) DIMENSION(*) for vslssseditrobustcovariance	Pointer to the array of method parameters

Name	Type	Description
	`REAL(KIND=8) DIMENSION(*)` for `vsldsseditrobustcovariance`	
rmean	`REAL(KIND=4) DIMENSION(*)` for `vslsssseditrobustcovariance` `REAL(KIND=8) DIMENSION(*)` for `vsldsseditrobustcovariance`	Pointer to the array of robust means
rcov	`REAL(KIND=4) DIMENSION(*)` for `vslsssseditrobustcovariance` `REAL(KIND=8) DIMENSION(*)` for `vsldsseditrobustcovariance`	Pointer to a robust covariance matrix

Output Parameters

Name	Type	Description
status	`INTEGER`	Current status of the task

Description

The `vslSSEditRobustCovariance` routine uses values passed as parameters of the routine to replace:

- pointers to covariance matrix storage
- pointers to the number of method parameters and to the array of the method parameters of size *nparams*
- pointers to the arrays that hold robust means and covariance

See Table "Storage formats of a variance-covariance/correlation matrix" for possible values of the *rcov_storage* parameter. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Intel® oneAPI Math Kernel Library (oneMKL) provides a Translated Biweight S-estimator (TBS) for robust estimation of a variance-covariance matrix and mean [Rocke96]. Use one iteration of the Maronna algorithm with the reweighting step [Maronna02] to compute the initial point of the algorithm. Pack the parameters of the TBS algorithm into the *params* array and pass them into the editor. Table "Structure of the Array of TBS Parameters" describes the *params* structure.

Structure of the Array of TBS Parameters

Array Position	Algorithm Parameter	Description
0	ε	Breakdown point, the number of outliers the algorithm can hold. By default, the value is $(n-p)/(2n)$.
1	α	Asymptotic rejection probability, see details in [Rocke96]. By default, the value is 0.001.
2	δ	Stopping criterion: the algorithm is terminated if weights are changed less than δ. By default, the value is 0.001.

Array Position	Algorithm Parameter	Description
3	max_iter	Maximum number of iterations. The algorithm terminates after max_iter iterations. By default, the value is 10. If you set this parameter to zero, the function returns a robust estimate of the variance-covariance matrix computed using the Maronna method [Maronna02] only.

The robust estimator of variance-covariance implementation in Intel® oneAPI Math Kernel Library (oneMKL) requires that the number of observations*n* be greater than twice the number of variables: $n > 2p$.

See additional details of the algorithm usage model in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [SS Notes].

vslSSEditOutliersDetection

Modifies array pointers related to multivariate outliers detection.

Syntax

```
status = vslssseditoutliersdetection(task, nparams, params, w)
status = vsldsseditoutliersdetection(task, nparams, params, w)
```

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
nparams	INTEGER	Pointer to the number of method parameters
params	REAL(KIND=4) DIMENSION(*) for vslssseditoutliersdetection REAL(KIND=8) DIMENSION(*) for vsldsseditoutliersdetection	Pointer to the array of method parameters
w	REAL(KIND=4) DIMENSION(*) for vslssseditoutliersdetection REAL(KIND=8) DIMENSION(*) for vsldsseditoutliersdetection	Pointer to an array of size *n*. The array holds the weights of observations to be marked as outliers.

Output Parameters

Name	Type	Description
status	INTEGER	Current status of the task

Description

The `vslSSEditOutliersDetection` routine uses the parameters passed to replace

- the pointers to the number of method parameters and to the array of the method parameters of size *nparams*
- the pointer to the array that holds the calculated weights of the observations

If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Intel® oneAPI Math Kernel Library (oneMKL) provides the BACON algorithm ([Billor00]) for the detection of multivariate outliers. Pack the parameters of the BACON algorithm into the *params* array and pass them into the editor. Table "Structure of the Array of BACON Parameters" describes the *params* structure.

Structure of the Array of BACON Parameters

Array Position	Algorithm Parameter	Description
0	Method to start the algorithm	The parameter takes one of the following possible values: `VSL_SS_METHOD_BACON_MEDIAN_INIT`, if the algorithm is started using the median estimate. This is the default value of the parameter. `VSL_SS_METHOD_BACON_MAHALANOBIS_INIT`, if the algorithm is started using the Mahalanobis distances.
1	α	One-tailed probability that defines the `(1 – α)` quantile of χ^2 distribution with `p` degrees of freedom. The recommended value is `α/ n`, where `n` is the number of observations. By default, the value is 0.05.
2	δ	Stopping criterion; the algorithm is terminated if the size of the basic subset is changed less than δ. By default, the value is 0.005.

Output of the algorithm is the vector of weights, `BaconWeights`, such that `BaconWeights`(*i*) = 0 if *i*-th observation is detected as an outlier. Otherwise `BaconWeights`(*i*) = *w*(*i*), where *w* is the vector of input weights. If you do not provide the vector of input weights, `BaconWeights`(*i*) is set to 1 if the *i*-th observation is not detected as an outlier.

See additional details about usage model of the algorithm in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [SS Notes].

vslSSEditMissingValues

Modifies pointers to arrays associated with the method of supporting missing values in a dataset.

Syntax

status = vslssseditmissingvalues(*task*, *nparams*, *params*, *init_estimates_n*, *init_estimates*, *prior_n*, *prior*, *simul_missing_vals_n*, *simul_missing_vals*, *estimates_n*, *estimates*)

status = vsldsseditmissingvalues(*task*, *nparams*, *params*, *init_estimates_n*, *init_estimates*, *prior_n*, *prior*, *simul_missing_vals_n*, *simul_missing_vals*, *estimates_n*, *estimates*)

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task
nparams	INTEGER	Pointer to the number of method parameters
params	REAL(KIND=4) DIMENSION(*) for vslsssseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues	Pointer to the array of method parameters
init_estimates_n	INTEGER	Pointer to the number of initial estimates for mean and a variance-covariance matrix
init_estimates	REAL(KIND=4) DIMENSION(*) for vslsssseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues	Pointer to the array that holds initial estimates for mean and a variance-covariance matrix
prior_n	INTEGER	Pointer to the number of prior parameters
prior	REAL(KIND=4) DIMENSION(*) for vslsssseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues	Pointer to the array of prior parameters
simul_missing_vals_n	INTEGER	Pointer to the size of the array that holds output of the Multiple Imputation method
simul_missing_vals	REAL(KIND=4) DIMENSION(*) for vslsssseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues	Pointer to the array of size *k*m*, where *k* is the total number of missing values, and *m* is number of copies of missing values. The array holds *m* sets of simulated missing values for the matrix of observations.
estimates_n	INTEGER	Pointer to the number of estimates to be returned by the routine
estimates	REAL(KIND=4) DIMENSION(*) for vslsssseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues	Pointer to the array that holds estimates of the mean and a variance-covariance matrix.

Output Parameters

Name	Type	Description
status	INTEGER	Current status of the task

Description

The vslSSEditMissingValues routine uses values passed as parameters of the routine to replace pointers to the number and the array of the method parameters, pointers to the number and the array of initial mean/variance-covariance estimates, the pointer to the number and the array of prior parameters, pointers to the number and the array of simulated missing values, and pointers to the number and the array of the intermediate mean/covariance estimates. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Before you call the Summary Statistics routines to process missing values, preprocess the dataset and denote missing observations with one of the following predefined constants:

- VSL_SS_SNAN, if the dataset is stored in single precision floating-point arithmetic
- VSL_SS_DNAN, if the dataset is stored in double precision floating-point arithmetic

Intel® oneAPI Math Kernel Library (oneMKL) provides theVSL_SS_METHOD_MI method to support missing values in the dataset based on the Multiple Imputation (MI) approach described in [Schafer97]. The following components support Multiple Imputation:

- Expectation Maximization (EM) algorithm to compute the start point for the Data Augmentation (DA) procedure
- DA function

> **NOTE**
> The DA component of the MI procedure is simulation-based and uses the VSL_BRNG_MCG59 basic random number generator with predefined *seed* = 2^{50} and the Gaussian distribution generator (ICDFmethod) available in Intel® oneAPI Math Kernel Library (oneMKL) [Gaussian].

Pack the parameters of the MI algorithm into the *params* array. Table "Structure of the Array of MI Parameters" describes the *params* structure.

Structure of the Array of MI Parameters

Array Position	Algorithm Parameter	Description
0	em_iter_num	Maximal number of iterations for the EM algorithm. By default, this value is 50.
1	da_iter_num	Maximal number of iterations for the DA algorithm. By default, this value is 30.
2	ε	Stopping criterion for the EM algorithm. The algorithm terminates if the maximal module of the element-wise difference between the previous and current parameter values is less than ε. By default, this value is 0.001.
3	m	Number of sets to impute
4	missing_vals_num	Total number of missing values in the datasets

You can also pass initial estimates into the EM algorithm by packing both the vector of means and the variance-covariance matrix as a one-dimensional array *init_estimates*. The size of the array should be at least $p + p(p + 1)/2$. For $i=0, .., p-1$, the *init_estimates*[*i*] array contains the initial estimate of means. The remaining positions of the array are occupied by the upper triangular part of the variance-covariance matrix.

If you provide no initial estimates for the EM algorithm, the editor uses the default values, that is, the vector of zero means and the unitary matrix as a variance-covariance matrix. You can also pass *prior* parameters for μ and Σ into the library: μ_0, τ, *m*, and Λ^{-1}. Pack these parameters as a one-dimensional array *prior* with a size of at least

$(p^2 + 3p + 4)/2$.

The storage format is as follows:

- *prior*[0], ..., *prior*[*p*–1] contain the elements of the vector μ_0.
- *prior*[*p*] contains the parameter τ.
- *prior*[*p*+1] contains the parameter *m*.
- The remaining positions are occupied by the upper-triangular part of the inverted matrix Λ^{-1}.

If you provide no *prior* parameters, the editor uses their default values:

- The array of *p* zeros is used as μ_0.
- τ is set to 0.
- *m* is set to *p*.
- The zero matrix is used as an initial approximate of Λ^{-1}.

The `EditMissingValues` editor returns *m* sets of imputed values and/or a sequence of parameter estimates drawn during the DA procedure.

The editor returns the imputed values as the *simul_missing_vals* array. The size of the array should be sufficient to hold *m* sets each of the `missing_vals_num` size, that is, at least *m**`missing_vals_num` in total. The editor packs the imputed values one by one in the order of their appearance in the matrix of observations.

For example, consider a task of dimension 4. The total number of observations *n* is 10. The second observation vector misses variables 1 and 2, and the seventh observation vector lacks variable 1. The number of sets to impute is *m*=2. Then, *simul_missing_vals*[0] and *simul_missing_vals*[1] contains the first and the second points for the second observation vector, and *simul_missing_vals*[2] holds the first point for the seventh observation. Positions 3, 4, and 5 are formed similarly.

To estimate convergence of the DA algorithm and choose a proper value of the number of DA iterations, request the sequence of parameter estimates that are produced during the DA procedure. The editor returns the sequence of parameters as a single array. The size of the array is

*m**da_iter_num*(*p*+(*p*^{2}+*p*)/2)

where

- *m* is the number of sets of values to impute.
- `da_iter_num` is the number of DA iterations.
- The value *p*+(*p*^{2}+*p*)/2 determines the size of the memory to hold one set of the parameter estimates.

In each set of the parameters, the vector of means occupies the first *p* positions and the remaining (*p*^{2}+*p*)/2 positions are intended for the upper triangular part of the variance-covariance matrix.

Upon successful generation of *m* sets of imputed values, you can place them in cells of the data matrix with missing values and use the Summary Statistics routines to analyze and get estimates for each of the *m* complete datasets.

NOTE
Intel® oneAPI Math Kernel Library (oneMKL) implementation of the MI algorithm rewrites cells of the dataset that contain the`VSL_SS_SNAN/VSL_SS_DNAN` values. If you want to use the Summary Statistics routines to process the data with missing values again, mask the positions of the empty cells.

See additional details of the algorithm usage model in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [SS Notes].

vslSSEditCorParameterization

Modifies pointers to arrays related to the algorithm of correlation matrix parameterization.

Syntax

```
status = vslsssseditcorparameterization(task, cor, cor_storage, pcor, pcor_storage)
status = vsldsseditcorparameterization(task, cor, cor_storage, pcor, pcor_storage)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
task	`TYPE(VSL_SS_TASK)`	Descriptor of the task
cor	`REAL(KIND=4) DIMENSION(*)` for `vslsssseditcorparameterization` `REAL(KIND=8) DIMENSION(*)` for `vsldsseditcorparameterization`	Pointer to the correlation matrix
cor_storage	`INTEGER`	Pointer to the storage format of the correlation matrix
pcor	`REAL(KIND=4) DIMENSION(*)` for `vslsssseditcorparameterization` `REAL(KIND=8) DIMENSION(*)` for `vsldsseditcorparameterization`	Pointer to the parameterized correlation matrix
por_storage	`INTEGER`	Pointer to the storage format of the parameterized correlation matrix

Output Parameters

Name	Type	Description
status	`INTEGER`	Current status of the task

Description

The `vslSSEditCorParameterization` routine uses values passed as parameters of the routine to replace pointers to the correlation matrix, pointers to the correlation matrix storage format, a pointer to the parameterized correlation matrix, and a pointer to the parameterized correlation matrix storage format. See

Table "Storage formats of a variance-covariance/correlation matrix" for possible values of the *cor_storage* and *pcor_storage* parameters. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Summary Statistics Task Computation Routines

Task computation routines calculate statistical estimates on the data provided and parameters held in the task descriptor. After you create the task and initialize its parameters, you can call the computation routines as many times as necessary. Table "Summary Statistics Estimates Obtained with vslSSCompute Routine" lists the respective statistical estimates.

NOTE
The Summary Statistics computation routines do not signal floating-point errors, such as overflow or gradual underflow, or operations with NaNs (except for the missing values in the observations).

Summary Statistics Estimates Obtained with vslSSCompute Routine

Estimate	Support of Observations Available in Blocks	Description
VSL_SS_MEAN	Yes	Computes the array of means.
VSL_SS_SUM	Yes	Computes the array of sums.
VSL_SS_2R_MOM	Yes	Computes the array of the 2nd order raw moments.
VSL_SS_2R_SUM	Yes	Computes the array of raw sums of the 2nd order.
VSL_SS_3R_MOM	Yes	Computes the array of the 3rd order raw moments.
VSL_SS_3R_SUM	Yes	Computes the array of raw sums of the 3rd order.
VSL_SS_4R_MOM	Yes	Computes the array of the 4th order raw moments.
VSL_SS_4R_SUM	Yes	Computes the array of raw sums of the 4th order.
VSL_SS_2C_MOM	Yes	Computes the array of the 2nd order central moments.
VSL_SS_2C_SUM	Yes	Computes the array of central sums of the 2nd order.
VSL_SS_3C_MOM	Yes	Computes the array of the 3rd order central moments.
VSL_SS_3C_SUM	Yes	Computes the array of central sums of the 3rd order.
VSL_SS_4C_MOM	Yes	Computes the array of the 4th order central moments.
VSL_SS_4C_SUM	Yes	Computes the array of central sums of the 4th order.

Estimate	Support of Observations Available in Blocks	Description
VSL_SS_KURTOSIS	Yes	Computes the array of kurtosis values.
VSL_SS_SKEWNESS	Yes	Computes the array of skewness values.
VSL_SS_MIN	Yes	Computes the array of minimum values.
VSL_SS_MAX	Yes	Computes the array of maximum values.
VSL_SS_VARIATION	Yes	Computes the array of variation coefficients.
VSL_SS_COV	Yes	Computes a covariance matrix.
VSL_SS_COR	Yes	Computes a correlation matrix. The main diagonal of the correlation matrix holds variances of the random vector components.
VSL_SS_CP	Yes	Computes a cross-product matrix.
VSL_SS_POOLED_COV	No	Computes a pooled covariance matrix.
VSL_SS_POOLED_MEAN	No	Computes an array of pooled means.
VSL_SS_GROUP_COV	No	Computes group covariance matrices.
VSL_SS_GROUP_MEAN	No	Computes group means.
VSL_SS_QUANTS	No	Computes quantiles.
VSL_SS_ORDER_STATS	No	Computes order statistics.
VSL_SS_ROBUST_COV	No	Computes a robust covariance matrix.
VSL_SS_OUTLIERS	No	Detects outliers in the dataset.
VSL_SS_PARTIAL_COV	No	Computes a partial covariance matrix.
VSL_SS_PARTIAL_COR	No	Computes a partial correlation matrix.
VSL_SS_MISSING_VALS	No	Supports missing values in datasets.
VSL_SS_PARAMTR_COR	No	Computes a parameterized correlation matrix.
VSL_SS_STREAM_QUANTS	Yes	Computes quantiles for streaming data.
VSL_SS_MDAD	No	Computes median absolute deviation.
VSL_SS_MNAD	No	Computes mean absolute deviation.
VSL_SS_SORTED_OBSERV	No	Sorts the dataset by the components of the random vector ξ.

Table "Summary Statistics Computation Method"lists estimate calculation methods supported by Intel® oneAPI Math Kernel Library (oneMKL). See the*Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [SS Notes] for a detailed description of the methods.

Summary Statistics Computation Method

Method	Description
VSL_SS_METHOD_FAST	Fast method for calculation of the estimates:

Method	Description
	• raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix • min/max/quantile/order statistics • partial variance-covariance • median/mean absolute deviation
VSL_SS_METHOD_FAST_USER_MEAN	Fast method for calculation of the estimates given user-defined mean: • central moments/sums of 2-4 order, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix, mean absolute deviation
VSL_SS_METHOD_1PASS	One-pass method for calculation of estimates: • raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix • pooled/group covariance matrix
VSL_SS_METHOD_TBS	TBS method for robust estimation of covariance and mean
VSL_SS_METHOD_BACON	BACON method for detection of multivariate outliers
VSL_SS_METHOD_MI	Multiple imputation method for support of missing values
VSL_SS_METHOD_SD	Spectral decomposition method for parameterization of a correlation matrix
VSL_SS_METHOD_SQUANTS_ZW	Zhang-Wang (ZW) method for quantile estimation for streaming data
VSL_SS_METHOD_SQUANTS_ZW_FAST	Fast ZW method for quantile estimation for streaming data
VSL_SS_METHOD_RADIX	Radix method for dataset sorting

You can calculate all requested estimates in one call of the routine. For example, to compute a kurtosis and covariance matrix using a fast method, pass a combination of the pre-defined parameters into the `Compute` routine as shown in the example below:

```
...
method = VSL_SS_METHOD_FAST;
task_params = VSL_SS_KURTOSIS|VSL_SS_COV;
…
status = vsldSSCompute( task, task_params, method );
```

To compute statistical estimates for the next block of observations, you can do one of the following:

- copy the observations to memory, starting with the address available to the task
- use one of the appropriate Editors to modify the pointer to the new dataset in the task.

The library does not detect your changes of the dataset and computed statistical estimates. To obtain statistical estimates for a new matrix, change the observations and initialize relevant arrays. You can follow this procedure to compute statistical estimates for observations that come in portions. See Table "Summary Statistics Estimates Obtained with `vslSSCompute` Routine"for information on such observations supported by the Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics estimators.

To modify parameters of the task using the Task Editors, set the address of the targeted matrix of the observations or change the respective vector component indices. After you complete editing the task parameters, you can compute statistical estimates in the modified environment.

If the task completes successfully, the computation routine returns the zero status code. If an error is detected, the computation routine returns an error code. In particular, an error status code is returned in the following cases:

- the task pointer is `NULL`
- memory allocation has failed
- the calculation has failed for some other reason

> **NOTE**
> You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vslSSCompute

Computes Summary Statistics estimates.

Syntax

```
status = vslssscompute(task, estimates, method)
status = vsldsscompute(task, estimates, method)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
task	`TYPE(VSL_SS_TASK)`	Descriptor of the task
estimates	`INTEGER (KIND=8)`	List of statistical estimates to compute
method	`INTEGER`	Method to be used in calculations

Output Parameters

Name	Type	Description
status	`INTEGER`	Current status of the task

Description

The `vslSSCompute` routine calculates statistical estimates passed as the *estimates* parameter using the algorithms passed as the *method* parameter of the routine. The computations are done in the context of the task descriptor that contains pointers to all required and optional, if necessary, properly initialized arrays. In one call of the function, you can compute several estimates using proper methods for their calculation. See Table "Summary Statistics Estimates Obtained with `Compute` Routine" for the list of the estimates that you can calculate with the `vslSSCompute` routine. See Table "Summary Statistics Computation Methods" for the list of possible values of the *method* parameter.

To initialize single or double precision version task parameters, use the single (`vslssscompute`) or double (`vsldsscompute`) version of the editor, respectively. To initialize parameters of the integer type, use an integer version of the editor (`vslisscompute`).

NOTE
Requesting a combination of the VSL_SS_MISSING_VALS value and any other estimate parameter in the Compute function results in processing only the missing values.

Application Notes

Be aware that when computing a correlation matrix, the vslSSCompute routine allocates an additional array for each thread which is running the task. If you are running on a large number of threads vslSSCompute might consume large amounts of memory.

When calculating covariance, correlation, or cross product, the number of bytes of memory required is at least (*P***P***T* + *P***T*)**b*, where *P* is the dimension of the task or number of variables, *T* is the number of threads, and *b* is the number of bytes required for each unit of data. If observation is weighted and the method is VSL_SS_METHOD_FAST, then the memory required is at least (*P***P***T* + *P***T* + *N***P*)**b*, where *N* is the number of observations.

Summary Statistics Task Destructor

Task destructor is the vslSSDeleteTask routine intended to delete task objects and release memory.

vslSSDeleteTask

Destroys the task object and releases the memory.

Syntax

status = vslssdeletetask(*task*)

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
task	TYPE(VSL_SS_TASK)	Descriptor of the task to destroy

Output Parameters

Name	Type	Description
status	INTEGER	Sets to VSL_STATUS_OK if the task is deleted; otherwise a non-zero code is returned.

Description

The vslSSDeleteTask routine deletes the task descriptor object, releases the memory allocated for the structure, and sets the task pointer to NULL. If vslSSDeleteTask fails to delete the task successfully, it returns an error code.

NOTE
Call of the destructor with the NULL pointer as the parameter results in termination of the function with no system crash.

Summary Statistics Usage Examples

The following examples show various standard operations with Summary Statistics routines.

Calculating Fixed Estimates for Fixed Data

The example shows recurrent calculation of the same estimates with a given set of variables for the complete life cycle of the task in the case of a variance-covariance matrix. The set of vector components to process remains unchanged, and the data comes in blocks. Before you call the `vslSSCompute` routine, initialize pointers to arrays for mean and covariance and set buffers.

```
….
double w[2];
double indices[DIM] = {1, 0, 1};

/* calculating mean for 1st and 3d random vector components */

/* Initialize parameters of the task */
p = DIM;
n = N;

xstorage   = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;

w[0] = 0.0; w[1] = 0.0;

for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;

status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, indices );

status = vsldSSEditTask  ( task, VSL_SS_ED_ACCUM_WEIGHT, w    );
status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );
```

You can process data arrays that come in blocks as follows:

```
for ( i = 0; i < num_of_blocks; i++ )
{
    status = vsldSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );
    /* Read new data block into array x */
}
…
```

Calculating Different Estimates for Variable Data

The context of your calculation may change in the process of data analysis. The example below shows the data that comes in two blocks. You need to estimate a covariance matrix for the complete data, and the third central moment for the second block of the data using the weights that were accumulated for the previous datasets. The second block of the data is stored in another array. You can proceed as follows:

```
/* Set parameters for the task */
p = DIM;
n = N;
xstorage   = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;

w[0] = 0.0; w[1] = 0.0;
```

```
for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;

/* Create task */
status = vsldSSNewTask( &task, &p, &n, &xstorage, x1, 0, indices );

/* Initialize the task parameters */
status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, w );
status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );

/* Calculate covariance for the x1 data */
status = vsldSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );

/* Initialize array of the 3d central moments and pass the pointer to the task */
for ( i = 0; i < p; i++ ) c3_m[i] = 0.0;

/* Modify task context */
status = vsldSSEditTask( task, VSL_SS_ED_3C_MOM, c3_m );
status = vsldSSEditTask( task, VSL_SS_ED_OBSERV, x2 );

/* Calculate covariance for the x1 & x2 data block */
/* Calculate the 3d central moment for the 2nd data block using earlier accumulated weight */
status = vsldSSCompute(task, VSL_SS_COV|VSL_SS_3C_MOM, VSL_SS_METHOD_FAST );
…
status = vslSSDeleteTask( &task );
```

Similarly, you can modify indices of the variables to be processed for the next data block.

Summary Statistics Mathematical Notation and Definitions

The following notations are used in the mathematical definitions and the description of the Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics functions.

Matrix and Weights of Observations

For a random *p*-dimensional vector $\xi = (\xi_1,..., \xi_i,..., \xi_p)$, this manual denotes the following:

- $(X)_i = (x_{ij})_{j=1..n}$ is the result of *n* independent observations for the *i*-th component ξ_i of the vector ξ.
- The two-dimensional array $X = (x_{ij})_{n \times p}$ is the matrix of observations.
- The column $[X]_j = (x_{ij})_{i=1..p}$ of the matrix *X* is the *j*-th observation of the random vector ξ.

Each observation $[X]_j$ is assigned a non-negative weight w_j, where

- The vector $(w_j)_{j=1..n}$ is a vector of weights corresponding to *n* observations of the random vector ξ.
- $W = \sum_{i=1}^{n} w_i$

  is the accumulated weight corresponding to observations *X*.

Vector of sample means

$$M(X) = (M_1(X), ..., M_p(X)) \text{ with } M_i(X) = \frac{1}{w} \sum_{j=1}^{n} w_j x_{ij}$$

for all $i = 1, ..., p$.

Vector of sample partial sums

$S(X) = \left(S_1(X), ..., S_p(X)\right)$ with $S_i(X) = \sum_{j=1}^{n} w_j x_{ij}$

for all *i* = 1, ..., *p*.

Vector of sample variances

$V(X) = \left(V_1(X), ..., V_p(X)\right)$ with $V_i(X) = \frac{1}{B}\sum_{j=1}^{n} w_j\left(x_{ij} - M_i(X)\right)^2$, $B = W - \sum_{j=1}^{n} w_j^2 / W$

for all *i* = 1, ..., *p*.

Vector of sample raw/algebraic moments of *k*-th order, *k*≥ 1

$R^{(k)}(X) = \left(R_1^{(k)}(X), ..., R_p^{(k)}(X)\right)$ with $R_i^{(k)}(X) = \frac{1}{W}\sum_{j=1}^{n} w_j x_{ij}^k$

for all *i* = 1, ..., *p*.

Vector of sample raw/algebraic partial sums of *k*-th order, *k*= 2, 3, 4 (raw/algebraic partial sums of squares/cubes/fourth powers)

$S^k(X) = \left(S_1^k(X), ..., S_p^k(X)\right)$ with $S_i^k(X) = \sum_{j=1}^{n} w_j x_{ij}^k$

for all *i* = 1, ..., *p*.

Vector of sample central moments of the third and the fourth order

$C^{(k)}(X) = \left(C_1^{(k)}(X), ..., C_p^{(k)}(X)\right)$ with $C_i^{(k)}(X) = \frac{1}{B}\sum_{j=1}^{n} w_j\left(x_{ij} - M_i(X)\right)^k$, $B = \sum_{j=1}^{n} w_j$

for all *i* = 1, ..., *p* and *k* = 3, 4.

Vector of sample central partial sums of *k*-th order, *k*= 2, 3, 4 (central partial sums of squares/cubes/fourth powers)

$S^k(X) = \left(S_1^k(X), ..., S_p^k(X)\right)$ with $S_i^k(X) = \sum_{j=1}^{n} w_j\left(x_{ij} - S_i(X)\right)^k$

for all *i* = 1, ..., *p*.

Vector of sample excess kurtosis values

$B(X) = \left(B_1(X), ..., B_p(X)\right)$ with $B_i(X) = \frac{C_i^{(4)}(X)}{V_i^2(X)} - 3$

for all *i* = 1, ..., *p*.

Vector of sample skewness values

$\Gamma(X) = \left(\Gamma_1(X), ..., \Gamma_p(X)\right)$ with $\Gamma_i(X) = \frac{C_i^{(3)}(X)}{V_i^{1.5}(X)}$

for all *i* = 1, ..., *p*.

Vector of sample variation coefficients

$VC(X) = \left(VC_1(X), ..., VC_p(X)\right)$ with $VC_i(X) = \frac{V_i^{0.5}(X)}{M_i(X)}$

for all *i* = 1, ..., *p*.

Matrix of order statistics

Matrix $Y = (y_{ij})_{pxn}$, in which the *i*-th row $(Y)_i = (y_{ij})_{j=1..n}$ is obtained as a result of sorting in the ascending order of row $(X)_i = (x_{ij})_{j=1..n}$ in the original matrix of observations.

Vector of sample minimum values

$Min(X) = \left(Min_1(X), ..., Min_p(X)\right)$, where $Min_i(X) = y_{i1}$

for all *i* = 1, ..., *p*.

Vector of sample maximum values

$Max(X) = \left(Max_1(X), ..., Max_p(X)\right)$, where $Max_i(X) = y_{in}$

for all *i* = 1, ..., *p*.

Vector of sample median values

$Med(X) = \left(Med_1(X), ..., Med_p(X)\right)$, where $Med_i(X) = \begin{cases} y_{i,(n+1)/2}, \text{if } n \text{ is odd} \\ \left(y_{i,n/2} + y_{i,n/2+1}\right)/2, \text{if } n \text{ is even} \end{cases}$

for all *i* = 1, ..., *p*.

Vector of sample median absolute deviations

$MDAD(X) = \left(MDAD_1(X), ..., MDAD_p(X)\right)$, where $MDAD_i(X) = Med_i(Z)$ with $Z = \left(z_{ij}\right)_{i=1...p, j=1...n}$, $z_{ij} = \left|x_{ij} - Med_i(X)\right|$

for all *i* = 1, ..., *p*.

Vector of sample mean absolute deviations

$MNAD(X) = \left(MNAD_1(X), ..., MNAD_p(X)\right)$, where $MNAD_i(X) = M_i(Z)$ with $Z = \left(z_{ij}\right)_{i=1...p, j=1...n}$, $z_{ij} = \left|x_{ij} - M_i(X)\right|$

for all *i* = 1, ..., *p*.

Vector of sample quantile values

For a positive integer number *q* and *k* belonging to the interval [0, *q*-1], point z_i is the *k*-th *q* quantile of the random variable ξ_i if $P\{\xi_i \le z_i\} \ge \beta$ and $P\{\xi_i \le z_i\} \ge 1 - \beta$, where

- *P* is the probability measure.
- $\beta = k/n$ is the quantile order.

The calculation of quantiles is as follows:

j = [(*n*-1)β] and *f* = {(*n*-1)β} as integer and fractional parts of the number (*n*-1)β, respectively, and the vector of sample quantile values is

$Q(X,\beta) = (Q_1(X,\beta), \ldots, Q_p(X,\beta))$

where

$(Q_i(X,\beta) = y_{i,j+1} + f(y_{i,j+2} - y_{i,j+1})$

for all *i* = 1, ..., *p*.

Variance-covariance matrix

$C(X) = (c_{ij}(X))_{p \times p}$

where

$$c_{ij}(X) = \frac{1}{B}\sum_{k=1}^{n} w_k\left(x_{ik} - M_i(X)\right)\left(x_{jk} - M_j(X)\right),\ B = W - \sum_{j=1}^{n} w_j^2 / W$$

Cross-product matrix (matrix of cross-products and sums of squares)

$CP(X) = (cp_{ij}(X))_{p \times p}$

where

$$cp_{ij}(X) = \sum_{k=1}^{n} w_k\left(x_{ik} - M_i(X)\right)\left(x_{jk} - M_j(X)\right)$$

Pooled and group variance-covariance matrices

The set *N* = {1, ..., *n*} is partitioned into non-intersecting subsets

$$G_i,\ i = 1..g,\ N = \bigcup_{i=1}^{g} G_i$$

The observation $[X]_j = (x_{ij})_{i=1..p}$ belongs to the group *r* if *j*∈*G*$_r$. One observation belongs to one group only. The group mean and variance-covariance matrices are calculated similarly to the formulas above:

$$M^{(r)}(X) = \left(M_1^{(r)}(X), \ldots, M_p^{(r)}(X)\right) \text{ with } M_i^{(r)}(X) = \frac{1}{W^{(r)}}\sum_{j \in G_r} w_j x_{ij},\ W^{(r)} = \sum_{j \in G_r} w_j$$

for all *i* = 1, ..., *p*,

$$C^{(r)}(X) = \left(c_{ij}^{(r)}(X)\right)_{p \times p}$$

where

$$c_{ij}^{(r)}(X) = \frac{1}{B^{(r)}}\sum_{k \in G_r} w_k\left(x_{ik} - M_i^{(r)}(X)\right)\left(x_{jk} - M_j^{(r)}(X)\right),\ B^{(r)} = W^{(r)} - \sum_{j \in G_r} w_j^2 / W^{(r)}$$

for all *i* = 1, ..., *p* and *j* = 1, ..., *p*.

A pooled variance-covariance matrix and a pooled mean are computed as weighted mean over group covariance matrices and group means, correspondingly:

$$M^{pooled}(X) = \left(M_1^{pooled}(X), \ldots, M_p^{pooled}(X)\right) \text{ with } M_i^{pooled}(X) = \frac{1}{W^{(1)} + \ldots + W^{(g)}}\sum_{r=1}^{g} W^{(r)} M_i^{(r)}(X)$$

for all *i* = 1, ..., *p*,

$C^{pooled}\left(X\right) = \left(c_{ij}^{pooled}(X)\right)_{p \times p}$, $c_{ij}^{pooled}(X) = \frac{1}{B^{(1)} + \ldots + B^{(g)}} \sum_{r=1}^{g} B^{(r)} c_{ij}^{(r)}(X)$

for all i = 1, ..., p and j = 1, ..., p.

Correlation matrix

$R(X) = \left(r_{ij}(X)\right)_{p \times p}$, where $r_{ij}\left(X\right) = \frac{c_{ij}}{\sqrt{c_{ii}c_{jj}}}$

for all i = 1, ..., p and j = 1, ..., p.

Partial variance-covariance matrix

For a random vector ξ partitioned into two components Z and Y, a variance-covariance matrix C describes the structure of dependencies in the vector ξ:

$$C(X) = \begin{pmatrix} C_Z(X) & C_{ZY}(X) \\ C_{YZ}(X) & C_Y(X) \end{pmatrix}.$$

The partial covariance matrix $P(X) = (p_{ij}(X))_{kxk}$ is defined as

$P\left(X\right) = C_Y\left(X\right) - C_{YZ}\left(X\right)C_Z^{-1}C_{ZY}(X).$

where k is the dimension of Y.

Partial correlation matrix

The following is a partial correlation matrix for all i = 1, ..., k and j = 1, ..., k:

$RP(X) = \left(rp_{ij}(X)\right)_{k \times k}$, where $rp_{ij}(X) = \frac{p_{ij}(X)}{\sqrt{p_{ii}(X)p_{jj}(X)}}$

where

- k is the dimension of Y.
- $p_{ij}(X)$ are elements of the partial variance-covariance matrix.

Sorted dataset

Matrix $Y = (y_{ij})_{pxn}$, in which the i-th row $(Y)_i$ is obtained as a result of sorting in ascending order the row $(X)_i = (x_{ij})_{j = 1..n}$ in the original matrix of observations.

Fourier Transform Functions

The general form of the discrete Fourier transform is

$$z_{k_1,\, k_2,\, \ldots,\, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \ldots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \ldots, j_d} \exp\left(\delta i 2\pi \sum_{l=1}^{d} j_l k_l / n_l\right)$$

for $k_l = 0, \ldots n_l-1$ (l = 1, ..., d), where σ is a scale factor, δ = -1 for the forward transform, and δ = +1 for the inverse (backward) transform. In the forward transform, the input (periodic) sequence $\{w_{j_1, j_2, \ldots, j_d}\}$ belongs to the set of complex-valued sequences and real-valued sequences. Respective domains for the backward transform are represented by complex-valued sequences and complex-valued conjugate-even sequences.

The Intel® oneAPI Math Kernel Library (oneMKL) provides an interface for computing a discrete Fourier transform through the fast Fourier transform algorithm. Prefixes `Dfti` in function names and `DFTI` in the names of configuration parameters stand for Discrete Fourier Transform Interface.

The manual describes the following implementations of the fast Fourier transform functions available in Intel® oneAPI Math Kernel Library (oneMKL):

- Fast Fourier transform (FFT) functions for single-processor or shared-memory systems (see FFT Functions)
- Cluster FFT functions for distributed-memory architectures (available only for Intel® 64 architectures)

NOTE
Intel® oneAPI Math Kernel Library (oneMKL) also supports the FFTW3* interfaces to the fast Fourier transform functionality for shared memory paradigm (SMP) systems.

Both FFT and Cluster FFT functions compute an FFT in five steps:

1. Allocate a fresh descriptor for the problem with a call to the `DftiCreateDescriptor` or `DftiCreateDescriptorDM` function. The descriptor captures the configuration of the transform, such as the dimensionality (or rank), sizes, number of transforms, memory layout of the input/output data (defined by strides), and scaling factors. Many of the configuration settings are assigned default values in this call which you might need to modify in your application.
2. Optionally adjust the descriptor configuration with a call to the `DftiSetValue` or `DftiSetValueDM` function as needed. Typically, you must carefully define the data storage layout for an FFT or the data distribution among processes for a Cluster FFT. The configuration settings of the descriptor, such as the default values, can be obtained with the `DftiGetValue` or `DftiGetValueDM` function.
3. Commit the descriptor with a call to the `DftiCommitDescriptor` or `DftiCommitDescriptorDM` function, that is, make the descriptor ready for the transform computation. Once the descriptor is committed, the parameters of the transform, such as the type and number of transforms, strides and distances, the type and storage layout of the data, and so on, are "frozen" in the descriptor.
4. Compute the transform with a call to the `DftiComputeForward`/`DftiComputeBackward` or `DftiComputeForwardDM`/`DftiComputeBackwardDM` functions as many times as needed. Because the descriptor is defined and committed separately, all that the compute functions do is take the input and output data and compute the transform as defined. To modify any configuration parameters for another call to a compute function, use `DftiSetValue` followed by `DftiCommitDescriptor` (`DftiSetValueDM` followed by `DftiCommitDescriptorDM`) or create and commit another descriptor.
5. Deallocate the descriptor with a call to the `DftiFreeDescriptor` or `DftiFreeDescriptorDM` function. This returns the memory internally consumed by the descriptor to the operating system.

All the above functions return an integer status value, which is zero upon successful completion of the operation. You can interpret a non-zero status with the help of the `DftiErrorClass` or `DftiErrorMessage` function.

The FFT functions support lengths with arbitrary factors. You can improve performance of the Intel® oneAPI Math Kernel Library (oneMKL) FFT if the length of your data vector permits factorization into powers of optimized radices. See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for specific radices supported efficiently.

NOTE
The FFT functions assume the Cartesian representation of complex data (that is, the real and imaginary parts define a complex number). The Intel® oneAPI Math Kernel Library (oneMKL) Vector Mathematical Functions provide efficient tools for conversion to and from polar representation (see Example "Conversion from Cartesian to polar representation of complex data" and Example "Conversion from polar to Cartesian representation of complex data").

FFT Functions

The fast Fourier transform function library of Intel® oneAPI Math Kernel Library (oneMKL) provides one-dimensional, two-dimensional, and multi-dimensional transforms (of up to seven dimensions) and offers both Fortran and C interfaces for all transform functions.

Table "FFT Functions in Intel® oneAPI Math Kernel Library (oneMKL)"lists FFT functions implemented in Intel® oneAPI Math Kernel Library (oneMKL):

FFT Functions in oneMKL

Function Name	Operation
Descriptor Manipulation Functions	
DftiCreateDescriptor	Allocates the descriptor data structure and initializes it with default configuration values.
DftiCommitDescriptor	Performs all initialization for the actual FFT computation.
DftiFreeDescriptor	Frees memory allocated for a descriptor.
DftiCopyDescriptor	Makes a copy of an existing descriptor.
FFT Computation Functions	
DftiComputeForward	Computes the forward FFT.
DftiComputeBackward	Computes the backward FFT.
Descriptor Configuration Functions	
DftiSetValue	Sets one particular configuration parameter with the specified configuration value.
DftiGetValue	Gets the value of one particular configuration parameter.
Status Checking Functions	
DftiErrorClass	Checks if the status reflects an error of a predefined class.
DftiErrorMessage	Translates the numeric value of an error status into a message.

FFT Interface

The Intel® oneAPI Math Kernel Library (oneMKL) FFT functions are provided with the Fortran and C interfaces.Fortran 95 is required because it offers features that have no counterpart in FORTRAN 77.

> **NOTE**
> The Fortran interface of the FFT computation functions requires one-dimensional data arrays for any dimension of FFT problem. For multidimensional transforms, pass the address of the first column of the multidimensional data to the computation functions.

To use the FFT functions, you need to access the module `MKL_DFTI` through the Fortran `use` statement.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR`, named constants representing various names of configuration parameters and their possible values, and overloaded functions through the generic functionality of Fortran 95.

NOTE
The current version of the library may not support some of the FFT functions or functionality. You can find the complete list of the implementation-specific exceptions in the Intel® oneAPI Math Kernel Library (oneMKL) Release Notes.

For the main categories of Intel® oneAPI Math Kernel Library (oneMKL) FFT functions, see FFT Functions.

Computing an FFT

You can find code examples that compute transforms in the Fourier Transform Functions Code Examples.

Usually you can compute an FFT by five function calls (refer to the usage model for details). A single data structure, the descriptor, stores configuration parameters that can be changed independently.

The descriptor data structure, when created, contains information about the length and domain of the FFT to be computed, as well as the setting of several configuration parameters. Default settings for some of these parameters are as follows:

- Scale factor: none (that is, $\sigma = 1$)
- Number of data sets: one
- Data storage: contiguous
- Placement of results: in-place (the computed result overwrites the input data)

The default settings can be changed one at a time through the function `DftiSetValue` as illustrated in Example "Changing Default Settings (Fortran)".

Configuration Settings

Each of the configuration parameters is identified by a named constant in the `MKL_DFTI` module.

All the Intel® oneAPI Math Kernel Library (oneMKL) FFT configuration parameters are readable. Some of them are read-only, while others can be set using the `DftiCreateDescriptor` or `DftiSetValue` function.

Values of the configuration parameters fall into the following groups:

- Values that have native data types. For example, the number of simultaneous transforms requested has an integer value, while the scale factor for a forward transform is a floating-point number.
- Values that are discrete in nature and are provided in the `MKL_DFTI` module as named constants. For example, the domain of the forward transform requires values to be named constants.

The Table "Configuration Parameters" summarizes the information on configuration parameters, along with their types and values. For more details of each configuration parameter, see the subsection describing this parameter.

Configuration Parameters

Configuration Parameter	Type/Value	Comments
Most common configuration parameters, no default, must be set explicitly by `DftiCreateDescriptor`		
`DFTI_PRECISION`	Named constant `DFTI_SINGLE` or `DFTI_DOUBLE`	Precision of the computation.
`DFTI_FORWARD_DOMAIN`	Named constant `DFTI_COMPLEX` or `DFTI_REAL`	Type of the transform.
`DFTI_DIMENSION`	Integer scalar	Dimension of the transform.
`DFTI_LENGTHS`	Integer scalar/array	Lengths of each dimension.
Common configuration parameters, settable by `DftiSetValue`		

Configuration Parameter	Type/Value	Comments
DFTI_PLACEMENT	Named constant DFTI_INPLACE or DFTI_NOT_INPLACE	Defines whether the result overwrites the input data. Default value: DFTI_INPLACE.
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor for the forward transform. Default value: 1.0. Precision of the value should be the same as defined by DFTI_PRECISION.
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor for the backward transform. Default value: 1.0. Precision of the value should be the same as defined by DFTI_PRECISION.
DFTI_NUMBER_OF_USER_THREADS	Integer scalar	This configuration parameter is no longer used and kept for compatibility with previous versions of Intel® oneAPI Math Kernel Library (oneMKL).
DFTI_THREAD_LIMIT	Integer scalar	Limits the number of threads for the DftiComputeForward and DftiComputeBackward. Default value: 0.
DFTI_DESCRIPTOR_NAME	Character string	Assigns a name to a descriptor. Assumed length of the string is DFTI_MAX_NAME_LENGTH. Default value: empty string.
Data layout configuration parameters for single and multiple transforms. Settable by DftiSetValue		
DFTI_INPUT_STRIDES	Integer array	Defines the input data layout. **NOTE** The default strides are set during creation of the descriptor based on the desired dimension and lengths. For more details, see DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES.
DFTI_OUTPUT_STRIDES	Integer array	Defines the output data layout. **NOTE** The default strides are set during creation of the descriptor based on the desired dimension and lengths. For more details, see DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES.
DFTI_NUMBER_OF_TRANSFORMS	Integer scalar	Number of transforms. Default value: 1.
DFTI_INPUT_DISTANCE	Integer scalar	Defines the distance between input data sets for multiple transforms. Default value: 0.

Configuration Parameter	Type/Value	Comments
DFTI_OUTPUT_DISTANCE	Integer scalar	Defines the distance between output data sets for multiple transforms. Default value: 0.
DFTI_COMPLEX_STORAGE	Named constant DFTI_COMPLEX_COMPLEX or DFTI_REAL_REAL	Defines whether the real and imaginary parts of data for a complex transform are interleaved in one array or split in two arrays. Default value: DFTI_COMPLEX_COMPLEX.
DFTI_REAL_STORAGE	Named constant DFTI_REAL_REAL	Defines how real data for a real transform is stored. Only the DFTI_REAL_REAL value is supported.
DFTI_CONJUGATE_EVEN_STORAGE	Named constant DFTI_COMPLEX_COMPLEX or DFTI_COMPLEX_REAL	Defines whether the complex data in the backward domain of a real transform is stored as complex elements or as real elements. DFTI_COMPLEX_REAL is supported only for 1D transforms. The default value is DFTI_COMPLEX_COMPLEX.
DFTI_PACKED_FORMAT	Named constant DFTI_CCE_FORMAT, DFTI_CCS_FORMAT, DFTI_PACK_FORMAT, or DFTI_PERM_FORMAT	Defines the layout for the elements of the conjugate-even sequence in the backward domain of the real transform (in association with the configuration parameter DFTI_CONJUGATE_EVEN_STORAGE). The default value is DFTI_CCE_FORMAT. **NOTE** Transforms greater than 1D support only DFTI_CCE_FORMAT.
Advanced configuration parameters, settable by DftiSetValue		
DFTI_WORKSPACE	Named constant DFTI_ALLOW or DFTI_AVOID	Defines whether the library should prefer algorithms using additional memory. Default value: DFTI_ALLOW.
DFTI_ORDERING	Named constant DFTI_ORDERED or DFTI_BACKWARD_SCRAMBLED	Defines whether the result of a complex transform is ordered or permuted. Default value: DFTI_ORDERED.
Read-Only configuration parameters		
DFTI_COMMIT_STATUS	Named constant DFTI_UNCOMMITTED or DFTI_COMMITTED	Readiness of the descriptor for computation.
DFTI_VERSION	String	Version of Intel® oneAPI Math Kernel Library (oneMKL). Assumed length of the string is DFTI_VERSION_LENGTH.

See Also

Configuring and Computing an FFT in Fortran

DFTI_PRECISION

The configuration parameter `DFTI_PRECISION` denotes the floating-point precision in which the transform is to be carried out. A setting of `DFTI_SINGLE` stands for single precision, and a setting of `DFTI_DOUBLE` stands for double precision. The data must be presented in this precision, the computation is carried out in this precision, and the result is delivered in this precision.

`DFTI_PRECISION` does not have a default value. Set it explicitly by calling the `DftiCreateDescriptor` function.

> **NOTE**
> Fortran module `MKL_DFTI` also defines named constants `DFTI_SINGLE_R` and `DFTI_DOUBLE_R`, with the same semantics as `DFTI_SINGLE` and `DFTI_DOUBLE`, respectively. Do not use these constants to set the `DFTI_PRECISION` configuration parameter. Use them only as described in `DftiCreateDescriptor`.

To better understand configuration of the precision of transforms, refer to these examples in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

```
./examples/dftf/source/basic_sp_complex_dft_1d.f90
```

```
./examples/dftf/source/basic_dp_complex_dft_1d.f90
```

See Also

`DFTI_FORWARD_DOMAIN`

`DFTI_DIMENSION, DFTI_LENGTHS`

`DftiCreateDescriptor`

DFTI_FORWARD_DOMAIN

The general form of a discrete Fourier transform is

$$z_{k_1,\,k_2,\ldots,\,k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \cdots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1,j_2,\ldots,\,j_d} \exp\left(\delta i 2\pi \sum_{l=1}^{d} j_l k_l / n_l\right)$$

for $k_l = 0, \ldots n_l-1$ (l = 1, ..., d), where σ is a scale factor, δ = -1 for the forward transform, and δ = +1 for the backward transform.

The Intel® oneAPI Math Kernel Library (oneMKL) implementation of the FFT algorithm, used for fast computation of discrete Fourier transforms, supports forward transforms on input sequences of two domains, as specified by the `DFTI_FORWARD_DOMAIN` configuration parameter: general complex-valued sequences (`DFTI_COMPLEX` domain) and general real-valued sequences (`DFTI_REAL` domain). The forward transform maps the forward domain to the corresponding backward domain, as shown in Table "Correspondence of Forward and Backward Domain".

The conjugate-even domain covers complex-valued sequences with the symmetry property:

$$x(k_1, k_2, \ldots, k_d) = conjugate(x(n_1 - k_1, n_2 - k_2, \ldots, n_d - k_d))$$

where the index arithmetic is performed modulo respective size, that is,

$$x(\ldots, expr_s, \ldots) \equiv x(\ldots, \mathrm{mod}(expr_s, n_s), \ldots),$$

and therefore

$$x(\ldots, n_s, \ldots) \equiv x(\ldots, 0, \ldots).$$

Due to this property of conjugate-even sequences, only a part of such sequence is stored in the computer memory, as described in `DFTI_CONJUGATE_EVEN_STORAGE`.

Correspondence of Forward and Backward Domain

Forward Domain	Implied Backward Domain
Complex (DFTI_COMPLEX)	Complex (DFTI_COMPLEX)
Real (DFTI_REAL)	Conjugate-even

DFTI_FORWARD_DOMAIN does not have a default value. Set it explicitly by calling the DftiCreateDescriptor function.

To better understand usage of the DFTI_FORWARD_DOMAIN configuration parameter, you can refer to these examples in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

./examples/dftf/source/basic_sp_complex_dft_1d.f90

./examples/dftf/source/basic_sp_real_dft_1d.f90

See Also

DFTI_PRECISION

DFTI_DIMENSION, DFTI_LENGTHS

DftiCreateDescriptor

DFTI_DIMENSION, DFTI_LENGTHS

The dimension of the transform is a positive integer value represented in an integer scalar of Integer data type. For a one-dimensional transform, the transform length is specified by a positive integer value represented in an integer scalar of Integer data type. For multi-dimensional (≥ 2) transform, the lengths of each of the dimensions are supplied in an integer array (of Integer data type).

DFTI_DIMENSION and DFTI_LENGTHS do not have a default value. To set them, use the DftiCreateDescriptor function and not the DftiSetValue function.

To better understand usage of the DFTI_DIMENSION and DFTI_LENGTHS configuration parameters, you can refer to basic examples of one-, two-, and three-dimensional transforms in your Intel® oneAPI Math Kernel Library (oneMKL) directory. Naming conventions for the examples are self-explanatory. For example, refer to these examples of single-precision two-dimensional transforms:

./examples/dftf/source/basic_sp_real_dft_2d.f90

./examples/dftf/source/basic_sp_complex_dft_2d.f90

See Also

DFTI_FORWARD_DOMAIN

DFTI_PRECISION

DftiCreateDescriptor

DftiSetValue

DFTI_PLACEMENT

By default, the computational functions overwrite the input data with the output result. That is, the default setting of the configuration parameter DFTI_PLACEMENT is DFTI_INPLACE. You can change that by setting it to DFTI_NOT_INPLACE.

> **NOTE**
> When the configuration parameter is set to DFTI_NOT_INPLACE, the input and output data sets must have no common elements.

To better understand usage of the DFTI_PLACEMENT configuration parameter, see this example in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

```
./examples/dftf/source/config_placement.f90
```

See Also

DftiSetValue

DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE

The forward transform and backward transform are each associated with a scale factor σ of its own having the default value of 1. You can specify the scale factors using one or both of the configuration parameters DFTI_FORWARD_SCALE and DFTI_BACKWARD_SCALE. For example, for a one-dimensional transform of length *n*, you can use the default scale of 1 for the forward transform and set the scale factor for the backward transform to be 1/*n*, thus making the backward transform the inverse of the forward transform.

Set the scale factor configuration parameter using a real floating-point data type of the same precision as the value for DFTI_PRECISION.

See Also

DftiSetValue

DFTI_PRECISION

DftiGetValue

DFTI_NUMBER_OF_USER_THREADS

The DFTI_NUMBER_OF_USER_THREADS configuration parameter is no longer used and kept for compatibility with previous versions of Intel® oneAPI Math Kernel Library (oneMKL).

See Also

DftiSetValue

DFTI_THREAD_LIMIT

In some situations you may need to limit the number of threads that the DftiComputeForward and DftiComputeBackward functions use. For example, if more than one thread calls Intel® oneAPI Math Kernel Library (oneMKL), it might be important that the thread calling these functions does not oversubscribe computing resources (CPU cores). Similarly, a known limit of the maximum number of threads to be used in computations might help the DftiCommitDescriptor function to select a more optimal computation method.

Set the parameter DFTI_THREAD_LIMIT as follows:

- To a positive number, to specify the maximum number of threads to be used by the compute functions.
- To zero (the default value), to use the maximum number of threads permitted in Intel® oneAPI Math Kernel Library (oneMKL) FFT functions. See "Techniques to Set the Number of Threads" in the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for more information.

On an attempt to set a negative value, the DftiSetValue function returns an error and does not update the descriptor.

The value of the DFTI_THREAD_LIMIT configuration parameter returned by the DftiGetValue function is defined as follows:

- 1 if Intel® oneAPI Math Kernel Library (oneMKL) runs in the sequential mode
- Depends of the commit status of the descriptor if Intel® oneAPI Math Kernel Library (oneMKL) runs in a threaded mode:

Commit Status	Value
Not committed	The value of DFTI_THREAD_LIMIT set in a previous call to the DftiSetValue function or the default value
Committed	The upper limit on the number of threads used by the DftiComputeForward and DftiComputeBackward functions

To better understand usage of the DFTI_THREAD_LIMIT configuration parameter, see this example in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

```
./examples/dftf/source/config_thread_limit.f90
```

See Also

DftiGetValue

DftiSetValue

DftiCommitDescriptor

DftiComputeForward

DftiComputeBackward

Threading Control Functions

DFTI_COMMIT_STATUS

DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES

The FFT interface provides configuration parameters that define the layout of multidimensional data in the computer memory. For *d*-dimensional data set *X* defined by dimensions N_1 x N_2 x ... x N_d, the layout describes where a particular element $X(k_1, k_2, ..., k_d)$ of the data set is located. The memory address of the element $X(k_1, k_2 , ..., k_d)$ is expressed by the formula,

address of $X(k_1, k_2, ..., k_d)$ = the address stored in the pointer supplied to the compute function + (s_0 + $k_1*s_1 + k_2*s_2 + ...+ k_d*s_d$) * u,

Where u is the number of bytes per element of the desired precision for the assumed data type in the corresponding domain (see Table "Assumed Element Types of the Input/Output Data" below), where s_0 is the displacement, and s_1, ..., s_d are generalized strides. The configuration parameters DFTI_INPUT_STRIDES and DFTI_OUTPUT_STRIDES enable you to get and set these values. The configuration value is an array of values (s_0, s_1, ..., s_d) of INTEGER data type.

The DFTI_FORWARD_DOMAIN, DFTI_COMPLEX_STORAGE, and DFTI_CONJUGATE_EVEN_STORAGE configuration parameters define the type of the elements as shown in Table "Assumed Element Types of the Input/Output Data":

Assumed Element Types of the Input/Output Data

Descriptor Configuration	Element Type in the Forward Domain	Element Type in the Backward Domain
DFTI_FORWARD_DOMAIN=DFTI_COMPLEX DFTI_COMPLEX_STORAGE=DFTI_COMPLEX_COMPLEX	Complex	Complex
DFTI_FORWARD_DOMAIN=DFTI_COMPLEX DFTI_COMPLEX_STORAGE=DFTI_REAL_REAL	Real	Real
DFTI_FORWARD_DOMAIN=DFTI_REAL DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL	Real	Real
DFTI_FORWARD_DOMAIN=DFTI_REAL DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX	Real	Complex

The DFTI_INPUT_STRIDES configuration parameter defines the layout of the input data, while the element type is defined by the forward domain for the DftiComputeForward function and by the backward domain for the DftiComputeBackward function. The DFTI_OUTPUT_STRIDES configuration parameter defines the layout of the output data, while the element type is defined by the backward domain for the DftiComputeForward function and by the forward domain for DftiComputeBackward function.

NOTE
The DFTI_INPUT_STRIDES and DFTI_OUTPUT_STRIDES configuration parameters define the layout of input and output data, and not the forward-domain and backward-domain data. If the data layouts in forward domain and backward domain differ, set DFTI_INPUT_STRIDES and DFTI_OUTPUT_STRIDES explicitly and then commit the descriptor before calling computation functions.

For in-place transforms (DFTI_PLACEMENT=DFTI_INPLACE), the configuration set by DFTI_OUTPUT_STRIDES is ignored when the element types in the forward and backward domains are the same. If they are different, set DFTI_OUTPUT_STRIDES explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the first elements on input and output must coincide in each dimension.

The FFT interface supports both positive and negative stride values. If you use negative strides, set the displacement of the data as follows:

$$s_0 = \sum_{i=1}^{d} (N_i - 1) \cdot \max(-s_i, 0).$$

The default setting of strides in a general multi-dimensional case assumes that the array that contains the data has no padding. The order of the strides depends on the programming language. For example:

```
INTEGER :: dims(d) = [n1, n2, …, nd]
status = DftiCreateDescriptor( hand, precision, domain, d, dims)
! The above call assumes data declaration:  type X(n1,n2,…,n_{d-1})
! Default strides are [ 0, 1, n1, n1*n2, …, n1*n2*…*n_{d-1}]
```

Note that in case of a real FFT (DFTI_FORWARD_DOMAIN=DFTI_REAL), where different data layouts in the backward domain are available (see DFTI_PACKED_FORMAT), the default value of the strides is not intuitive for the recommended CCE format (configuration setting DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX). In case of an *in-place* real transform with the CCE format, set the strides explicitly, as follows:

```
INTEGER :: dims(d) = [n1, n2, …, nd]
INTEGER :: rstrides(1+d) = [0, 1, 2*(n1/2+1), 2*(n1/2+1)*n2, …, 2*n_{d-1}*…*n_2*(n_1/2+1)]
INTEGER :: cstrides(1+d) = [0, 1, (n1/2+1), (n1/2+1)*n2, …, n_{d-1}*…*n_2*(n_1/2+1) ]
status = DftiCreateDescriptor( hand, precision, domain, d, dims)
status = DftiSetValue( hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)
! Set the strides appropriately for forward/backward transform
```

Limitation Note Transforms with the number of points N of a non-unit stride dimension exceeding 2^(27−p) −1 for N a power-of-two, or 2^(23−p) − 1 for N not a power-of-two, are currently not supported, where p=0 for single precision and p=1 for double precision. If a descriptor is created (for example, using DftiCreateDescriptor) and set (for example, using DftiSetValue) to do such a transform, a DFTI_1D_MEMORY_EXCEEDS_INT32 error is returned at commit time (for example, by DftiCommitDescriptor).

To better understand configuration of strides, you can also refer to these examples in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

./examples/dftf/source/basic_sp_complex_dft_2d.f90

./examples/dftf/source/basic_sp_complex_dft_3d.f90

./examples/dftf/source/basic_dp_complex_dft_2d.f90

./examples/dftf/source/basic_dp_complex_dft_3d.f90

./examples/dftf/source/basic_sp_real_dft_2d.f90

```
./examples/dftf/source/basic_sp_real_dft_3d.f90
./examples/dftf/source/basic_dp_real_dft_2d.f90
./examples/dftf/source/basic_dp_real_dft_3d.f90
```

See Also

DFTI_FORWARD_DOMAIN

DFTI_PLACEMENT

FFT Code Examples

DftiSetValue

DftiCommitDescriptor

DftiComputeForward

DftiComputeBackward

DFTI_NUMBER_OF_TRANSFORMS

If you need to perform a large number of identical FFTs, you can do this in a single call to a DftiCompute* function with the value of the DFTI_NUMBER_OF_TRANSFORMS configuration parameter equal to the actual number of the transforms. The default value of this parameter is 1. You can set this parameter to a positive integer value of the Integer data type. When setting the number of transforms to a value greater than one, you also need to specify the distance between the input data sets and the distance between the output data sets using one of the DFTI_INPUT_DISTANCE and DFTI_OUTPUT_DISTANCE configuration parameters or both.

Important

- The data sets to be transformed must not have common elements.
- All the sets of data must be located within the same memory block.

To better understand usage of the DFTI_NUMBER_OF_TRANSFORMS configuration parameter, see this example in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

```
./examples/dftf/source/config_number_of_transforms.f90
```

See Also

FFT Computation Functions

DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE

DftiSetValue

DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE

The FFT interface in Intel® oneAPI Math Kernel Library (oneMKL) enables computation of multiple transforms. To compute multiple transforms, you need to specify the data distribution of the multiple sets of data. The distance between the first data elements of consecutive data sets, DFTI_INPUT_DISTANCE for input data or DFTI_OUTPUT_DISTANCE for output data, specifies the distribution. The configuration setting is a value of INTEGER data type.

The default value for both configuration settings is one. You must set this parameter explicitly if the number of transforms is greater than one (see DFTI_NUMBER_OF_TRANSFORMS).

The distance is counted in elements of the data type defined by the descriptor configuration (rather than by the type of the variable passed to the computation functions). Specifically, the DFTI_FORWARD_DOMAIN, DFTI_COMPLEX_STORAGE, and DFTI_CONJUGATE_EVEN_STORAGE configuration parameters define the type of the elements as shown in Table "Assumed Element Types of the Input/Output Data".

NOTE
The configuration parameters DFTI_INPUT_DISTANCE and DFTI_OUTPUT_DISTANCE define the distance within input and output data, and not within the forward-domain and backward-domain data. If the distances in the forward and backward domains differ, set DFTI_INPUT_DISTANCE and DFTI_OUTPUT_DISTANCE explicitly and then commit the descriptor before calling computation functions.

For in-place transforms (DFTI_PLACEMENT=DFTI_INPLACE), the configuration set by DFTI_OUTPUT_DISTANCE is ignored when the element types in the forward and backward domains are the same. If they are different, set DFTI_OUTPUT_DISTANCE explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the data sets on input and output must coincide.

This example illustrates setting of the DFTI_INPUT_DISTANCE configuration parameter:

```
INTEGER :: dims(d) = [n1, n2, …, nd]
INTEGER :: distance = n1*n2*…*nd
status = DftiCreateDescriptor( hand, precision, DFTI_COMPLEX, d, dims)
status = DftiSetValue( hand, DFTI_NUMBER_OF_TRANSFORMS, howmany )
status = DftiSetValue( hand, DFTI_INPUT_DISTANCE,  distance );
```

To better understand configuration of the distances, see these code examples in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

./examples/dftf/source/config_number_of_transforms.f90

See Also

DFTI_PLACEMENT

DftiSetValue

DftiCommitDescriptor

DftiComputeForward

DftiComputeBackward

DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE

Depending on the value of the DFTI_FORWARD_DOMAIN configuration parameter, the implementation of FFT supports several storage schemes for input and output data (see document [3] for the rationale behind the definition of the storage schemes). The data elements are placed within contiguous memory blocks, defined with generalized strides (see DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES). For multiple transforms, all sets of data should be located within the same memory block, and the data sets should be placed at the same distance from each other (see DFTI_NUMBER_OF_TRANSFORMS and DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE).

FFT Examples demonstrate the usage of storage formats.

DFTI_COMPLEX_STORAGE: storage schemes for a complex domain

For the DFTI_COMPLEX forward domain, both input and output sequences belong to a complex domain. In this case, the configuration parameter DFTI_COMPLEX_STORAGE can have one of the two values: DFTI_COMPLEX_COMPLEX (default) or DFTI_REAL_REAL.

NOTE
In the Intel® oneAPI Math Kernel Library (oneMKL)FFT interface, storage schemes for a forward complex domain and the respective backward complex domain are the same.

With DFTI_COMPLEX_COMPLEX storage, complex-valued data sequences are referenced by a single complex parameter (array) AZ so that a complex-valued element $z_{k_1, k_2, ..., k_d}$ of the m-th d-dimensional sequence is located at AZ[m*distance + stride$_0$ + k_1*stride$_1$ + k_2*stride$_2$+ ... k_d*stride$_d$] as a structure consisting of the real and imaginary parts.

This code illustrates the use of the DFTI_COMPLEX_COMPLEX storage:

```
complex :: AZ(N1,N2,N3,M)   ! sizes and number of transforms
...
! on input:  Z{k1,k2,k3,m} = AZ(k1,k2,k3,m)
status = DftiComputeForward( desc, AZ(:,1,1,1) )
! on output: Z{k1,k2,k3,m} = AZ(k1,k2,k3,m)
```

With the DFTI_REAL_REAL storage, complex-valued data sequences are referenced by two real parameters AR and AI so that a complex-valued element $z_{k_1, k_2, ..., k_d}$ of the m-th sequence is computed as AR[m*distance + stride$_0$ + k_1*stride$_1$ + k_2*stride$_2$+ ... k_d*stride$_d$] + √(-1) * AI[m*distance + stride$_0$ + k_1*stride$_1$ + k_2*stride$_2$+ ... k_d*stride$_d$].

This code illustrates the use of the DFTI_REAL_REAL storage:

```
real :: AR(N1,N2,N3,M), AI(N1,N2,N3,M)
...
! on input:  Z{k1,k2,k3,m} = cmplx(AR(k1,k2,k3,m),AI(k1,k2,k3,m))
status = DftiComputeForward( desc, AR(:,1,1,1), AI(:,1,1,1) )
! on output: Z{k1,k2,k3,m} = cmplx(AR(k1,k2,k3,m),AI(k1,k2,k3,m))
```

DFTI_REAL_STORAGE: storage schemes for a real domain

The Intel® oneAPI Math Kernel Library (oneMKL) FFT interface supports only one configuration value for this storage scheme: DFTI_REAL_REAL. With the DFTI_REAL_REAL storage, real-valued data sequences in a real domain are referenced by one real parameter AR so that real-valued element of the m-th sequence is located as AR[m*distance + stride$_0$ + k_1*stride$_1$ + k_2*stride$_2$+ ... k_d*stride$_d$].

DFTI_CONJUGATE_EVEN_STORAGE: storage scheme for a conjugate-even domain

The Intel® oneAPI Math Kernel Library (oneMKL) FFT interface supports two configuration values for this parameter: DFTI_COMPLEX_COMPLEX (default) and DFTI_COMPLEX_REAL (for 1D problems only). The conjugate-even symmetry of the data enables storing only about a half of the whole mathematical result, so that one part of it can be directly referenced in the memory while the other part can be reconstructed depending on the selected storage configuration.

With the DFTI_COMPLEX_COMPLEX storage, the complex-valued data sequences in the conjugate-even domain are referenced by one complex parameter AZ so that a complex-valued element $z_{k_1, k_2, ..., k_d}$ of the m-th sequence can be referenced or reconstructed as described below.

Consider a *d*-dimensional real-to-complex transform:

$$Z_{k_1,k_2,\ldots,k_d} \equiv \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} R_{n_1,n_2,\ldots,n_d} e^{\frac{-2\pi i}{N_1} k_1 \cdot n_1} \cdots e^{\frac{-2\pi i}{N_d} k_d \cdot n_d}$$

Because the input sequence *R* is real-valued, the mathematical result *Z* has conjugate-even symmetry:

$z_{k_1, k_2, ..., k_d}$ = *conjugate* ($z_{N_1-k_1, N_2-k_2, ..., N_d-k_d}$),

where index arithmetic is performed modulo the length of the respective dimension. Obviously, the first element of the result is real-valued:

$z_{0, 0, ..., 0}$ = *conjugate* ($z_{0, 0, ..., 0}$).

For dimensions with even lengths, some of the other elements are real-valued too. For example, if N_s is even, $z_{0, 0, ..., N_s/2, 0, ..., 0} = conjugate\ (z_{0, 0, ..., N_s/2, 0, ..., 0})$.

With the conjugate-even symmetry, approximately a half of the result suffices to fully reconstruct it. For an arbitrary dimension h, it suffices to store elements $z_{k_1, ..., k_h, ..., k_d}$ for the following indices:

- $k_h = 0, ..., \lfloor N_h/2 \rfloor$
- $k_i = 0, ..., N_i - 1$, where $i = 1, ..., d$ and $i \neq h$

The symmetry property enables reconstructing the remaining elements: for $k_h = \lfloor N_h/2 \rfloor + 1, ..., N_h - 1$. In the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface, the halved dimension is the first dimension.

The following code illustrates usage of the DFTI_COMPLEX_COMPLEX storage for a conjugate-even domain:

```
real :: AR(N1,N2,M)          ! Array containing values of R
complex :: AZ(N1/2+1,N2,M)   ! Array containing values of Z
...
! on input:  R{k1,k2,m} = AR(k1,k2,m)
status = DftiComputeForward( desc, AR(:,1,1), AZ(:,1,1) )
! on output:
! for k1=1 … N1/2+1:  Z{k1,k2,m} = AZ(k1,k2,m)
! for k1=N1/2+2 … N1: Z{k1,k2,m} = conj(AZ(mod(N1-k1+1,N1)+1,mod(N2-k2+1,N2)+1,m))
```

For the backward transform, the input and output parameters and layouts exchange roles: set the strides describing the layout in the backward/forward domain as input/output strides, respectively. For example:

```
...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES,  fwd_domain_strides )
status = DftiSetValue( desc, DFTI_OUTPUT_STRIDES, bwd_domain_strides )
status = DftiCommitDescriptor( desc )
status = DftiComputeForward( desc, ... )
...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES,  bwd_domain_strides )
status = DftiSetValue( desc, DFTI_OUTPUT_STRIDES, fwd_domain_strides )
status = DftiCommitDescriptor( desc )
status = DftiComputeBackward( desc, ... )
```

Important
For in-place transforms, ensure the first element of the input data has the same location as the first element of the output data for each dimension.

See Also

DftiSetValue

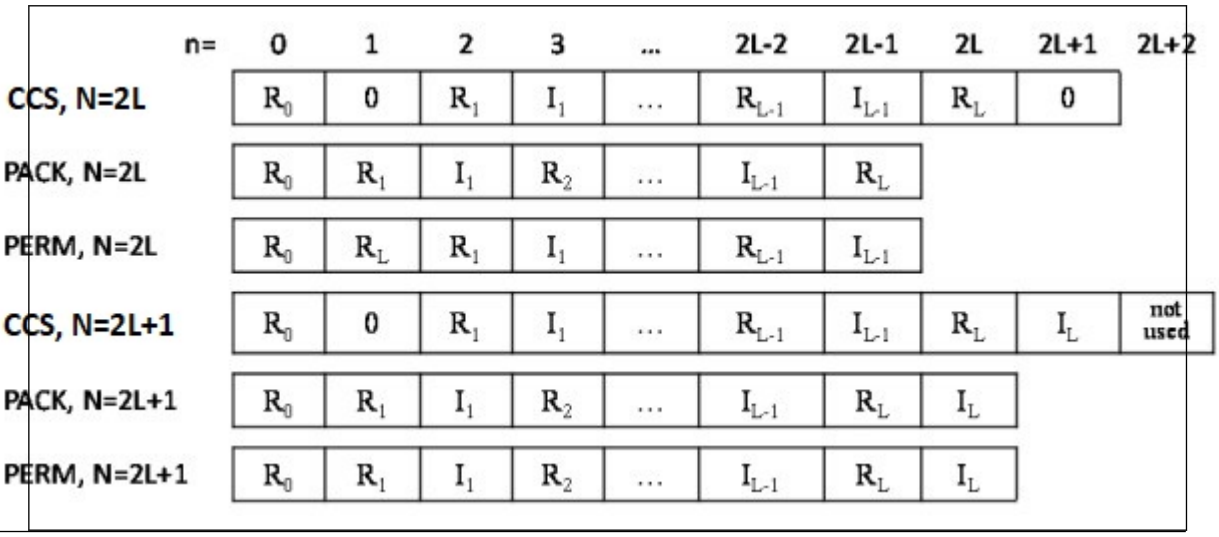
DFTI_PACKED_FORMAT

The result of the forward transform of real data is a conjugate-even sequence. Due to the symmetry property, only a part of the complex-valued sequence is stored in memory. The combination of the DFTI_PACKED_FORMAT and DFTI_CONJUGATE_EVEN_STORAGE configuration parameters defines how the conjugate-even sequence data is packed. If DFTI_CONJUGATE_EVEN_STORAGE is set to DFTI_COMPLEX_COMPLEX (default), the only possible value of DFTI_PACKED_FORMAT is DFTI_CCE_FORMAT; this association of configuration parameters is supported for transforms of any dimension. For a description of the corresponding packed format, see DFTI_CONJUGATE_EVEN_STORAGE. For one-dimensional transforms (only) with DFTI_CONJUGATE_EVEN_STORAGE set to DFTI_COMPLEX_REAL, the DFTI_PACKED_FORMAT configuration parameter must be DFTI_CCS_FORMAT, DFTI_PACK_FORMAT, or DFTI_PERM_FORMAT. The corresponding packed formats are explained and illustrated below.

DFTI_CCS_FORMAT for One-dimensional Transforms

The following figure illustrates the storage of a one-dimensional (1D) size-*N* conjugate-even sequence in a real array for the CCS, PACK, and PERM packed formats. The CCS format requires an array of size *N*+2, while the other formats require an array of size *N*. Zero-based indexing is used.

Storage of a 1D Size-*N* Conjugate-even Sequence in a Real Array



NOTE For storage of a one-dimensional conjugate-even sequence in a real array, CCS is in the same format as CCE.

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in a real-valued array `AC` as illustrated by figure "Storage of a 1D Size-*N* Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```
real :: AR(N), AC(N+2)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT )
...
! on input:  R{k} = AR(k)
status = DftiComputeForward( desc, AR, AC )  ! real-to-complex FFT
! on output:
! for k=1 … N/2+1: Z{k} = cmplx( AC(1 + (2*(k-1)+0)),
!                                AC(1 + (2*(k-1)+1)) )
! for k=N/2+2 … N: Z{k} = cmplx( AC(1 + (2*mod(N-k+1,N)+0)),
!                               -AC(1 + (2*mod(N-k+1,N)+1)))
```

DFTI_PACK_FORMAT for One-dimensional Transforms

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in a real-valued array `AC` as illustrated by figure "Storage of a 1D Size-*N* Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```
real :: AR(N), AC(N)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT )
...
! on input:  R{k} = AR(k)
status = DftiComputeForward( desc, AR, AC )  ! real-to-complex FFT
! on output: Z{k} = cmplx( re, im ), where
```

```
!  if (k == 1) then
!    re =  AC(1)
!    im =  0
!  else if (k-1 == N-k+1) then
!    re =  AC(2*(k-1))
!    im =  0
!  else if (k <= N/2+1) then
!    re =  AC(2*(k-1)+0)
!    im =  AC(2*(k-1)+1)
!  else
!    re =  AC(2*(N-k+1)+0)
!    im = -AC(2*(N-k+1)+1)
!  end if
```

DFTI_PERM_FORMAT for One-dimensional Transforms

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in real-valued array `AC` as illustrated by figure "Storage of a 1D Size-*N* Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```
real :: AR(N), AC(N)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT )
...
! on input:  R{k} = AR(k)
status = DftiComputeForward( desc, AR, AC )  ! real-to-complex FFT
! on output: Z{k} = cmplx( re, im ), where
! if (k == 1) then
!   re =  AC(1)
!   im =  0
! else if (k-1 == N-k+1) then
!   re =  AC(2)
!   im =  0
! else if (k <= N/2+1) then
!   re =  AC(1+2*(k-1)+0-mod(N,2))
!   im =  AC(1+2*(k-1)+1-mod(N,2))
! else
!   re =  AC(1+2*(N-k+1)+0-mod(N,2))
!   im = -AC(1+2*(N-k+1)+1-mod(N,2))
! end if
```

See Also

`DftiSetValue`

DFTI_WORKSPACE

The computation step for some FFT algorithms requires a scratch space for permutation or other purposes. To manage the use of the auxiliary storage, Intel® oneAPI Math Kernel Library (oneMKL) enables you to set the configuration parameter`DFTI_WORKSPACE` with the following values:

`DFTI_ALLOW`	(default) Permits the use of the auxiliary storage.
`DFTI_AVOID`	Instructs Intel® oneAPI Math Kernel Library (oneMKL) to avoid using the auxiliary storage if possible.

See Also

`DftiSetValue`

DFTI_COMMIT_STATUS

The DFTI_COMMIT_STATUS configuration parameter indicates whether the descriptor is ready for computation. The parameter has two possible values:

DFTI_UNCOMMITTED	Default value, set after a successful call of DftiCreateDescriptor.
DFTI_COMMITTED	The value after a successful call to DftiCommitDescriptor.

A computation function called with an uncommitted descriptor returns an error.

You cannot directly set this configuration parameter in a call to DftiSetValue, but a change in the configuration of a committed descriptor may change the commit status of the descriptor to DFTI_UNCOMMITTED.

See Also

DftiCreateDescriptor

DftiCommitDescriptor

DftiSetValue

DFTI_ORDERING

Some FFT algorithms apply an explicit permutation stage that is time consuming [4]. The exclusion of this step is similar to applying an FFT to input data whose order is scrambled, or allowing a scrambled order of the FFT results. In applications such as convolution and power spectrum calculation, the order of result or data is unimportant and thus using scrambled data is acceptable if it leads to better performance. The following options are available in Intel® oneAPI Math Kernel Library (oneMKL):

- DFTI_ORDERED: Forward transform data ordered, backward transform data ordered (default option).
- DFTI_BACKWARD_SCRAMBLED: Forward transform data ordered, backward transform data scrambled.

Table "Scrambled Order Transform" tabulates the effect of this configuration setting.

Scrambled Order Transform

	DftiComputeForward	**DftiComputeBackward**
DFTI_ORDERING	Input → Output	Input → Output
DFTI_ORDERED	ordered → ordered	ordered → ordered
DFTI_BACKWARD_SCRAMBLED	ordered → scrambled	scrambled → ordered

NOTE
The word "scrambled" in this table means "permit scrambled order if possible". In some situations permitting out-of-order data gives no performance advantage and an implementation may choose to ignore the suggestion.

See Also

DftiSetValue

FFT Descriptor Manipulation Functions

This category contains the following functions: create a descriptor, commit a descriptor, copy a descriptor, and free a descriptor.

DftiCreateDescriptor

Allocates the descriptor data structure and initializes it with default configuration values.

Syntax

```
status = DftiCreateDescriptor( desc_handle, precision, forward_domain, dimension,
length )
```

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
precision	INTEGER	Precision of the transform: DFTI_SINGLE or DFTI_DOUBLE.
forward_domain	INTEGER	Forward domain of the transform: DFTI_COMPLEX or DFTI_REAL.
dimension	INTEGER	Dimension of the transform.
length	INTEGER if *dimension* = 1. Array INTEGER, DIMENSION(*) otherwise.	Length of the transform for a one-dimensional transform. Lengths of each dimension for a multi-dimensional transform.

Output Parameters

Name	Type	Description
desc_handle	DFTI_DESCRIPTOR	FFT descriptor.
status	INTEGER	Function completion status.

Description

This function allocates memory for the descriptor data structure and instantiates it with all the default configuration settings for the precision, forward domain, dimension, and length of the desired transform. Because memory is allocated dynamically, the result is actually a pointer to the created descriptor. This function is slightly different from the "initialization" function that can be found in software packages or libraries that implement more traditional algorithms for computing an FFT. This function does not perform any significant computational work such as computation of twiddle factors. The function DftiCommitDescriptor does this work after the function DftiSetValue has set values of all necessary parameters.

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Interface

```
                ! Note that the body provided below only illustrates the list of different
! parameters and the types of dummy parameters. You can rely only on the function
! name following keyword INTERFACE. For the precise definition of the
! interface, see the include/mkl_dfti.f90 file in the Intel MKL directory.

  INTERFACE DftiCreateDescriptor
```

```
    FUNCTION some_actual_function_1d(desc, precision, domain, dim, length)
      INTEGER :: some_actual_function_1d
      ...
      INTEGER, INTENT(IN) :: length
    END FUNCTION some_actual_function_1d

    FUNCTION some_actual_function_md(desc, precision, domain, dim, lengths)
      INTEGER :: some_actual_function_md
      ...
      INTEGER, INTENT(IN), DIMENSION(*) :: lengths
    END FUNCTION some_actual_function_md

    ...

END INTERFACE DftiCreateDescriptor
```

Note that the function is overloaded: the actual parameter for the formal parameter *length* can be a scalar or a rank-one array.

The function is also overloaded with respect to the type of the *precision* parameter in order to provide an option of using a precision-specific function for the generic name. Using more specific functions can reduce the size of statically linked executable for the applications using only single-precision FFTs or only double-precision FFTs. To use specific functions, change the "USE MKL_DFTI" statement in your program unit to one of the following:

```
USE MKL_DFTI, FORGET=>DFTI_SINGLE, DFTI_SINGLE=>DFTI_SINGLE_R
```

```
USE MKL_DFTI, FORGET=>DFTI_DOUBLE, DFTI_DOUBLE=>DFTI_DOUBLE_R
```

where the name "FORGET" can be replaced with any name that is not used in the program unit.

See Also

DFTI_PRECISION configuration parameter
DFTI_FORWARD_DOMAIN configuration parameter
DFTI_DIMENSION, DFTI_LENGTHS configuration parameters
Configuration Parameters, summary table

DftiCommitDescriptor

Performs all initialization for the actual FFT computation.

Syntax

status = DftiCommitDescriptor(*desc_handle*)

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
desc_handle	DFTI_DESCRIPTOR	FFT descriptor.

Output Parameters

Name	Type	Description
desc_handle	DFTI_DESCRIPTOR	Updated FFT descriptor.
status	INTEGER	Function completion status.

Description

This function completes initialization of a previously created descriptor, which is required before the descriptor can be used for FFT computations. Typically, committing the descriptor performs all initialization that is required for the actual FFT computation. The initialization done by the function may involve exploring different factorizations of the input length to find the optimal computation method.

If you call the DftiSetValue function to change configuration parameters of a committed descriptor (see Descriptor Configuration Functions), you must re-commit the descriptor before invoking a computation function. Typically, a committal function call is immediately followed by a computation function call (see FFT Computation Functions).

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Interface

```
INTERFACE DftiCommitDescriptor
!Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the
!keyword INTERFACE
 FUNCTION some_actual function_1 ( Desc_Handle )
 INTEGER :: some_actual function_1
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 END FUNCTION some_actual function_1
END INTERFACE DftiCommitDescriptor
```

DftiFreeDescriptor

Frees the memory allocated for a descriptor.

Syntax

status = DftiFreeDescriptor(*desc_handle*)

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
desc_handle	DESCRIPTOR_HANDLE	FFT descriptor.

Output Parameters

Name	Type	Description
desc_handle	DESCRIPTOR_HANDLE	Memory for the FFT descriptor is released.
status	INTEGER	Function completion status.

Description

This function frees all memory allocated for a descriptor.

> **NOTE**
> Memory allocation/deallocation inside Intel® oneAPI Math Kernel Library (oneMKL) is managed by Intel® oneAPI Math Kernel Library (oneMKL) memory management software. So, even after successful completion of `FreeDescriptor`, the memory space may continue being allocated for the application because the memory management software sometimes does not return the memory space to the OS, but considers the space free and can reuse it for future memory allocation. See Example mkl_free_buffers: Usage with FFT Functions on how to use Intel® oneAPI Math Kernel Library (oneMKL) memory management software and release memory to the OS.

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Interface

```
INTERFACE DftiFreeDescriptor
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_3( Desc_Handle )
 INTEGER :: some_actual_function_3
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 END FUNCTION some_actual_function_3
END INTERFACE DftiFreeDescriptor
```

DftiCopyDescriptor

Makes a copy of an existing descriptor.

Syntax

status = DftiCopyDescriptor(*desc_handle_original*, *desc_handle_copy*)

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
desc_handle_original	DESCRIPTOR_HANDLE	The FFT descriptor to copy.
	DESCRIPTOR_HANDLE	The FFT descriptor to copy.

Output Parameters

Name	Type	Description
dfti_handle_dst	DESCRIPTOR_HANDLE	The copy of the FFT descriptor.
status	INTEGER	Function completion status.

Description

This function makes a copy of an existing descriptor. The resulting descriptor *desc_handle_copy* and the existing descriptor *desc_handle_original* specify the same configuration of the transform, but do not have any memory areas in common ("deep copy").

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Interface

```
INTERFACE DftiCopyDescriptor
! Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the
!keyword INTERFACE
 FUNCTION some_actual_function_2( Desc_Handle_Original,
 Desc_Handle_Copy )
 INTEGER :: some_actual_function_2
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Original, Desc_Handle_Copy
 END FUNCTION some_actual_function_2
END INTERFACE DftiCopyDescriptor
```

FFT Descriptor Configuration Functions

This category contains the following functions: the value setting function DftiSetValue sets one particular configuration parameter to an appropriate value, and the value-getting function DftiGetValue reads the value of one particular configuration parameter. While all configuration parameters are readable, you cannot set a few of them. Some of these contain fixed information of a particular implementation such as version number, or dynamic information, which is derived by the implementation during execution of one of the functions. See Configuration Settings for details.

DftiSetValue

Sets one particular configuration parameter with the specified configuration value.

Syntax

status = DftiSetValue(*desc_handle*, *config_param*, *config_val*)

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
desc_handle	DFTI_DESCRIPTOR	FFT descriptor.
config_param	INTEGER	Configuration parameter.
config_val	Depends on the configuration parameter.	Configuration value.

Output Parameters

Name	Type	Description
desc_handle	DFTI_DESCRIPTOR	Updated FFT descriptor.
status	INTEGER	Function completion status.

Description

This function sets one particular configuration parameter with the specified configuration value. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- DFTI_PRECISION
- DFTI_FORWARD_DOMAIN
- DFTI_DIMENSION, DFTI_LENGTHS
- DFTI_PLACEMENT
- DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE
- DFTI_THREAD_LIMIT
- DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES
- DFTI_NUMBER_OF_TRANSFORMS
- DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE
- DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE
- DFTI_PACKED_FORMAT
- DFTI_WORKSPACE
- DFTI_ORDERING

You cannot use the DftiSetValue function to change configuration parameters DFTI_FORWARD_DOMAIN, DFTI_PRECISION, DFTI_DIMENSION, and DFTI_LENGTHS. Use the DftiCreateDescriptor function to set them.

Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in Configuring and Computing an FFT in Fortran .

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Interface

```
                INTERFACE DftiSetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_6_INTVAL( Desc_Handle, Config_Param, INTVAL )
 INTEGER :: some_actual_function_6_INTVAL
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 INTEGER, INTENT(IN) :: INTVAL
 END FUNCTION some_actual_function_6_INTVAL
 FUNCTION some_actual_function_6_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
 INTEGER :: some_actual_function_6_SGLVAL
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 REAL, INTENT(IN) :: SGLVAL
END FUNCTION some_actual_function_6_SGLVAL
 FUNCTION some_actual_function_6_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
 INTEGER :: some_actual_function_6_DBLVAL
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 REAL (KIND(0D0)), INTENT(IN) :: DBLVAL
 END FUNCTION some_actual_function_6_DBLVAL
 FUNCTION some_actual_function_6_INTVEC( Desc_Handle, Config_Param, INTVEC )
 INTEGER :: some_actual_function_6_INTVEC
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 INTEGER, INTENT(IN) :: INTVEC(*)
 END FUNCTION some_actual_function_6_INTVEC
 FUNCTION some_actual_function_6_CHARS( Desc_Handle, Config_Param, CHARS )
 INTEGER :: some_actual_function_6_CHARS
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 CHARCTER(*), INTENT(IN) :: CHARS
 END FUNCTION some_actual_function_6_CHARS
END INTERFACE DftiSetValue
```

See Also

Configuration Settings for more information on configuration parameters.

DftiCreateDescriptor

DftiGetValue

DftiGetValue

Gets the configuration value of one particular configuration parameter.

Syntax

status = DftiGetValue(*desc_handle*, *config_param*, *config_val*)

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
desc_handle	DFTI_DESCRIPTOR	FFT descriptor.
config_param	INTEGER	Configuration parameter. See Table "Configuration Parameters" for allowable values of *config_param*.

Output Parameters

Name	Type	Description
config_val	Depends on the configuration parameter.	Configuration value.
status	INTEGER	Function completion status.

Description

This function gets the configuration value of one particular configuration parameter. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- DFTI_PRECISION
- DFTI_FORWARD_DOMAIN
- DFTI_DIMENSION, DFTI_LENGTH
- DFTI_PLACEMENT
- DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE
- DFTI_THREAD_LIMIT
- DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES
- DFTI_NUMBER_OF_TRANSFORMS
- DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE
- DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE
- DFTI_PACKED_FORMAT
- DFTI_WORKSPACE
- DFTI_COMMIT_STATUS
- DFTI_ORDERING

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Interface

```
               INTERFACE DftiGetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_7_INTVAL( Desc_Handle, Config_Param, INTVAL )
 INTEGER :: some_actual_function_7_INTVAL
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
```

```
INTEGER, INTENT(OUT) :: INTVAL
END FUNCTION DFTI_GET_VALUE_INTVAL
FUNCTION some_actual_function_7_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
INTEGER :: some_actual_function_7_SGLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL, INTENT(OUT) :: SGLVAL
END FUNCTION some_actual_function_7_SGLVAL
FUNCTION some_actual_function_7_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
INTEGER :: some_actual_function_7_DBLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL (KIND(0D0)), INTENT(OUT) :: DBLVAL
END FUNCTION some_actual_function_7_DBLVAL
FUNCTION some_actual_function_7_INTVEC( Desc_Handle, Config_Param, INTVEC )
INTEGER :: some_actual_function_7_INTVEC
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(OUT) :: INTVEC(*)
END FUNCTION some_actual_function_7_INTVEC
FUNCTION some_actual_function_7_INTPNT( Desc_Handle, Config_Param, INTPNT )
INTEGER :: some_actual_function_7_INTPNT
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, DIMENSION(*), POINTER :: INTPNT
END FUNCTION some_actual_function_7_INTPNT
FUNCTION some_actual_function_7_CHARS( Desc_Handle, Config_Param, CHARS )
INTEGER :: some_actual_function_7_CHARS
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
CHARCTER(*), INTENT(OUT):: CHARS
END FUNCTION some_actual_function_7_CHARS
END INTERFACE DftiGetValue
```

Configuration Settings for more information on configuration parameters.

DftiSetValue

FFT Computation Functions

This category contains the following functions: compute the forward transform and compute the backward transform.

DftiComputeForward

Computes the forward FFT.

Syntax

status = DftiComputeForward(*desc_handle*, *x_inout*)

status = DftiComputeForward(*desc_handle*, *x_in*, *y_out*)

status = DftiComputeForward(*desc_handle*, *xre_inout*, *xim_inout*)

status = DftiComputeForward(*desc_handle*, *xre_in*, *xim_in*, *yre_out*, *yim_out*)

Input Parameters

Name	Type	Description
desc_handle	DFTI_DESCRIPTOR	FFT descriptor.
x_inout, x_in	Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Data to be transformed in case of a real forward domain or, in the case of a complex forward domain in association with FTI_COMPLEX_COMPLEX, set for DFTI_COMPLEX_STORAGE.
xre_inout, xim_inout, xre_in, xim_in	Array REAL(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the data to be transformed in the case of a complex forward domain.

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- *_inout* to DFTI_INPLACE
- *_in* to DFTI_NOT_INPLACE

Output Parameters

Name	Type	Description
y_out	Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	The transformed data in case of a real backward domain or, in the case of a complex forward domain in association with DFTI_COMPLEX_COMPLEX, set for DFTI_COMPLEX_STORAGE.
xre_inout, xim_inout, yre_out, yim_out	Array REAL(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the transformed data in the case of a complex forward domain in association with DFTI_REAL_REAL set for DFTI_COMPLEX_STORAGE.
status	INTEGER	Function completion status.

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- *_inout* to DFTI_INPLACE
- *_out* to DFTI_NOT_INPLACE

Include Files

- mkl_dfti.f90

Description

The DftiComputeForward function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, this function computes the forward FFT, that is, the transform with the minus sign in the exponent, $\delta = -1$.

The DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, and DFTI_CONJUGATE_EVEN_STORAGE configuration parameters define the layout of the input and output data and must be properly set in a call to the DftiSetValue function. The forward domain and the precision of the transform are determined by the configuration settings DFTI_FORWARD_DOMAIN and DFTI_PRECISION, which are during construction of the descriptor.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in Configuring and Computing an FFT in Fortran.

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by the generic interface. The generic Fortran interface to the computation functions is based on a set of specific functions. These functions can check for inconsistency between the required and actual number of parameters. However, the specific functions disregard the type of the actual parameters and instead use the interpretation defined in the descriptor by configuration parameters DFTI_FORWARD_DOMAIN, DFTI_INPUT_STRIDES, DFTI_INPUT_DISTANCE, and so on.

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Interface

```
! Note that the body provided below only illustrates the list of different
! parameters and the types of dummy parameters. You can rely only on the function
! name following keyword INTERFACE. For the precise definition of the
! interface, see the include/mkl_dfti.f90 file in the Intel MKL directory.
INTERFACE DftiComputeForward

  FUNCTION some_actual_function_1(desc,sSrcDst)
    INTEGER some_actual_function_1
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDst
    ...
  END FUNCTION some_actual_function_1

  FUNCTION some_actual_function_2(desc,cSrcDst)
    INTEGER some_actual_function_2
    COMPLEX(8), INTENT(INOUT), DIMENSION(*) :: cSrcDst
    ...
  END FUNCTION some_actual_function_2

  FUNCTION some_actual_function_3(desc,sSrcDstRe,sSrcDstIm)
    INTEGER some_actual_function_3
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstRe
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstIm
    ...
  END FUNCTION some_actual_function_3
  ...
END INTERFACE DftiComputeForward
```

The Fortran interface requires that the data parameters have the type of assumed-size rank-1 array, even for multidimensional transforms. The implementations of the FFT interface require the data stored linearly in memory with a regular stride pattern capable of describing multidimensional array layout (see also [3] and the more detailed discussion in DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES), and the function requires that the data parameters refer to the first element of the data. Consequently, the data arrays should be specified with the DIMENSION(*) attribute and the storage associated with the actual multidimensional arrays via the EQUIVALENCE statement.

See Also

Configuration Settings
DFTI_FORWARD_DOMAIN
DFTI_PLACEMENT
DFTI_PACKED_FORMAT
DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE
DFTI_DIMENSION, DFTI_LENGTHS
DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE
DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES
DftiComputeBackward
DftiSetValue

DftiComputeBackward

Computes the backward FFT.

Syntax

```
status = DftiComputeBackward( desc_handle, x_inout )
status = DftiComputeBackward( desc_handle, y_in, x_out )
status = DftiComputeBackward( desc_handle, xre_inout, xim_inout )
status = DftiComputeBackward( desc_handle, yre_in, yim_in, xre_out, xim_out )
```

Input Parameters

Name	Type	Description
desc_handle	DFTI_DESCRIPTOR	FFT descriptor.
x_inout, y_in	Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Data to be transformed in case of a real forward domain or, in the case of a complex forward domain in association with DFTI_COMPLEX_COMPLEX, set for DFTI_COMPLEX_STORAGE.
xre_inout, xim_inout, yre_in, yim_in	Array REAL(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the data to be transformed in the case of a complex forward domain in association with DFTI_REAL_REAL set for DFTI_COMPLEX_STORAGE.

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- *_inout* to DFTI_INPLACE
- *_in* to DFTI_NOT_INPLACE

Output Parameters

Name	Type	Description
x_out	Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	The transformed data in case of a real forward domain or, in the case of a complex forward domain in association with DFTI_COMPLEX_COMPLEX, set for DFTI_COMPLEX_STORAGE.
xre_inout, *xim_inout*, *xre_out*, *xim_out*	Array REAL(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the transformed data in the case of a complex forward domain in association with DFTI_REAL_REAL set for DFTI_COMPLEX_STORAGE.
status	INTEGER	Function completion status.

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- *_inout* to DFTI_INPLACE
- *_out* to DFTI_NOT_INPLACE

Include Files

- mkl_dfti.f90

Description

The function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, the DftiComputeBackward function computes the inverse FFT, that is, the transform with the plus sign in the exponent, $\delta = +1$.

The DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, and DFTI_CONJUGATE_EVEN_STORAGE configuration parameters define the layout of the input and output data and must be properly set in a call to the DftiSetValue function. The forward domain and the precision of the transform are determined by the configuration settings DFTI_FORWARD_DOMAIN and DFTI_PRECISION, which are during construction of the descriptor.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in Configuring and Computing an FFT in Fortran.

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by the generic interface. The generic Fortran interface to the computation functions is based on a set of specific functions. These functions can check for inconsistency between the required and actual number of parameters. However, the specific functions disregard the type of the actual parameters and instead use the interpretation defined in the descriptor by configuration parameters DFTI_FORWARD_DOMAIN, DFTI_INPUT_STRIDES, DFTI_INPUT_DISTANCE, and so on.

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Interface

```
! Note that the body provided below only illustrates the list of different
! parameters and the types of dummy parameters. You can rely only on the function
```

```
! name following keyword INTERFACE. For the precise definition of the
! interface, see the include/mkl_dfti.f90 file in the Intel MKL directory.
INTERFACE DftiComputeBackward

  FUNCTION some_actual_function_1(desc,sSrcDst)
    INTEGER some_actual_function_1
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDst
    ...
  END FUNCTION some_actual_function_1

  FUNCTION some_actual_function_2(desc,cSrcDst)
    INTEGER some_actual_function_2
    COMPLEX(8), INTENT(INOUT), DIMENSION(*) :: cSrcDst
    ...
  END FUNCTION some_actual_function_2

  FUNCTION some_actual_function_3(desc,sSrcDstRe,sSrcDstIm)
    INTEGER some_actual_function_3
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstRe
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstIm
    ...
  END FUNCTION some_actual_function_3

  ...
END INTERFACE DftiComputeBackward
```

The Fortran interface requires that the data parameters have the type of assumed-size rank-1 array, even for multidimensional transforms. The implementations of the FFT interface require the data stored linearly in memory with a regular stride pattern capable of describing multidimensional array layout (see also [3] and the more detailed discussion in `DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES`), and the function requires that the data parameters refer to the first element of the data. Consequently, the data arrays should be specified with the `DIMENSION(*)` attribute and the storage associated with the actual multidimensional arrays via the `EQUIVALENCE` statement.

See Also

Configuration Settings

`DFTI_FORWARD_DOMAIN`

`DFTI_PLACEMENT`

`DFTI_PACKED_FORMAT`

`DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE`

`DFTI_DIMENSION, DFTI_LENGTHS`

`DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE`

`DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES`

`DftiComputeForward`

`DftiSetValue`

Configuring and Computing an FFT in Fortran

The table below summarizes information on configuring and computing an FFT in Fortran for all kinds of transforms and possible combinations of input and output domains.

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
Complex-to-complex, in-place, forward or backward	Interleaved complex numbers	Interleaved complex numbers	! Configure a Descriptor status = DftiCreateDescriptor(hand, <*precision*>, & DFTI_COMPLEX, <*dimension*>, <*sizes*>) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, X_inout) ! or backward FFT status = DftiComputeBackward(hand, X_inout)
Complex-to-complex, out-of-place, forward or backward	Interleaved complex numbers	Interleaved complex numbers	! Configure a Descriptor status = DftiCreateDescriptor(hand, <*precision*>, & DFTI_COMPLEX, <*dimension*>, <*sizes*>) status = DftiSetValue(hand, DFTI_PLACEMENT, & DFTI_NOT_INPLACE) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, X_in, Y_out) ! or backward FFT status = DftiComputeBackward(hand, X_in, Y_out)
Complex-to-complex, in-place, forward or backward	Split-complex numbers	Split-complex numbers	! Configure a Descriptor status = DftiCreateDescriptor(hand, <*precision*>, & DFTI_COMPLEX, <*dimension*>, <*sizes*>) status = DftiSetValue(hand, & DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, Xre_inout, & Xim_inout) ! or backward FFT status = DftiComputeBackward(hand, Xre_inout, & Xim_inout)
Complex-to-complex, out-of-place, forward or backward	Split-complex numbers	Split-complex numbers	! Configure a Descriptor status = DftiCreateDescriptor(hand, <*precision*>, & DFTI_COMPLEX, <*dimension*>, <*sizes*>) status = DftiSetValue(hand, & DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL) status = DftiSetValue(hand, & DFTI_PLACEMENT, DFTI_NOT_INPLACE) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT

FFT to Compute	Input Data	Output Data	Required FFT Function Calls

			`status = DftiComputeForward(hand, Xre_in, &`
			`  Xim_in, Yre_out, Yim_out)`
			`! or backward FFT`
			`status = DftiComputeBackward(hand, Xre_in, &`
			`  Xim_in, Yre_out, Yim_out)`

Real-to-complex, in-place, forward	Real numbers	Numbers in the CCE format	`! Configure a Descriptor`
			`status = DftiCreateDescriptor(hand,`
			`<precision>, &`
			`  DFTI_REAL, <dimension>, <sizes>)`
			`status = DftiSetValue(hand, &`
			`  DFTI_CONJUGATE_EVEN_STORAGE, &`
			`  DFTI_COMPLEX_COMPLEX)`
			`status = DftiSetValue(hand,`
			`DFTI_PACKED_FORMAT, &`
			`  DFTI_CCE_FORMAT)`
			`status = DftiSetValue(hand,`
			`DFTI_INPUT_STRIDES, &`
			`  <real_strides>)`
			`status = DftiSetValue(hand,`
			`DFTI_OUTPUT_STRIDES, &`
			`  <complex_strides>)`
			`status = DftiCommitDescriptor(hand)`
			
			`! Compute an FFT`
			`status = DftiComputeForward(hand, X_inout)`

Real-to-complex, out-of-place, forward	Real numbers	Numbers in the CCE format	`! Configure a Descriptor`
			`status = DftiCreateDescriptor(hand, <precision>`
			`&,`
			`  DFTI_REAL, <dimension>, <sizes>)`
			`status = DftiSetValue(hand, &`
			`  DFTI_CONJUGATE_EVEN_STORAGE, &`
			`  DFTI_COMPLEX_COMPLEX)`
			`status = DftiSetValue(hand, DFTI_PACKED_FORMAT,`
			`&`
			`  DFTI_CCE_FORMAT)`
			`status = DftiSetValue(hand, DFTI_PLACEMENT, &`
			`  DFTI_NOT_INPLACE)`
			`status = DftiSetValue(hand, DFTI_INPUT_STRIDES,`
			`&`
			`  <real_strides>)`
			`status = DftiSetValue(hand,`
			`DFTI_OUTPUT_STRIDES, &`
			`  <complex_strides>)`
			`status = DftiCommitDescriptor(hand)`
			
			`! Compute an FFT`
			`status = DftiComputeForward(hand, X_in, Y_out)`

Complex-to-real, in-place, backward	Numbers in the CCE format	Real numbers	`! Configure a Descriptor`
			`status = DftiCreateDescriptor(hand,`
			`<precision>, &`
			`  DFTI_REAL, <dimension>, <sizes>)`

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
			status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <*complex_strides*>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <*real_strides*>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeBackward(hand, X_inout)
Complex-to-real, out-of-place, backward	Numbers in the CCE format	Real numbers	! Configure a Descriptor status = DftiCreateDescriptor(hand, <*precision*>, & DFTI_REAL, <*dimension*>, <*sizes*>) status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PLACEMENT, & DFTI_NOT_INPLACE) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <*complex_strides*>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <*real_strides*>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeBackward(hand, X_in, Y_out)

You can find Fortran programs that illustrate configuring and computing FFTs in the `examples/dftf/` subdirectory of your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Status Checking Functions

All of the descriptor manipulation, FFT computation, and descriptor configuration functions return an integer value denoting the status of the operation. The functions in this category check that status. The first function is a logical function that checks whether the status reflects an error of a predefined class, and the second is an error message function that returns a character string.

DftiErrorClass

Checks whether the status reflects an error of a predefined class.

Syntax

predicate = DftiErrorClass(*status*, *error_class*)

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
status	INTEGER	Completion status of a fast Fourier transform (FFT) function.
error_class	INTEGER	Predefined error class.

Output Parameters

Name	Type	Description
predicate	LOGICAL	Result of checking.

Description

The FFT interface in Intel® oneAPI Math Kernel Library (oneMKL) provides a set of predefined error classeslisted in Table "Predefined Error Classes". They are named constants and have the type INTEGER.

Predefined Error Classes

Named Constants	Comments
DFTI_NO_ERROR	No error. The zero status belongs to this class.
DFTI_MEMORY_ERROR	Usually associated with memory allocation.
DFTI_INVALID_CONFIGURATION	Invalid settings in one or more configuration parameters.
DFTI_INCONSISTENT_CONFIGURATION	Inconsistent configuration or input parameters.
DFTI_NUMBER_OF_THREADS_ERROR	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function).
DFTI_MULTITHREADED_ERROR	Usually associated with a value that OMP routines return in case of errors.
DFTI_BAD_DESCRIPTOR	Descriptor is unusable for computation.
DFTI_UNIMPLEMENTED	Unimplemented legitimate settings; implementation dependent.
DFTI_MKL_INTERNAL_ERROR	Internal library error.
DFTI_1D_LENGTH_EXCEEDS_INT32	Length of one of the dimensions exceeds 2^{32} -1 (4 bytes).
DFTI_1D_MEMORY_EXCEEDS_INT32	Data size of one of the transforms exceeds 2^{31} -1 bytes.

NOTE
Use `DFTI_1D_MEMORY_EXCEEDS_INT32` instead of `DFTI_1D_LENGTH_EXCEEDS_INT32` for better accuracy.

The `DftiErrorClass` function returns the value of `.TRUE.` if the status belongs to the predefined error class. To check whether a function call was successful, call `DftiErrorClass` with a specific error class. However, the zero value of the status belongs to the `DFTI_NO_ERROR` class and thus the zero status indicates successful completion of an operation. See Example "Using Status Checking Functions" for an illustration of correct use of the status checking functions.

NOTE
It is incorrect to directly compare a status with a predefined class.

Interface

```
INTERFACE DftiErrorClass
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_8( Status, Error_Class )
 LOGICAL some_actual_function_8
 INTEGER, INTENT(IN) :: Status, Error_Class
 END FUNCTION some_actual_function_8
END INTERFACE DftiErrorClass
```

DftiErrorMessage

Generates an error message.

Syntax

error_message = DftiErrorMessage(*status*)

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
status	INTEGER	Completion status of a function.

Output Parameters

Name	Type	Description
error_message	CHARACTER(LEN=DFTI_MAX_MESSAGE_LENGTH)	The character string with the error message.

Description

The error message function generates an error message character string. In Fortran, use a character string of length `DFTI_MAX_MESSAGE_LENGTH` as a target for the error message.

Example Using Status Checking Function shows how this function can be used.

Interface

```
INTERFACE DftiErrorMessage
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_9( Status )
 CHARACTER(LEN=DFTI_MAX_MESSAGE_LENGTH) some_actual_function_9( Status )
 INTEGER, INTENT(IN) :: Status
 END FUNCTION some_actual_function_9
END INTERFACE DftiErrorMessage
```

Cluster FFT Functions

This section describes the cluster Fast Fourier Transform (FFT) functions implemented in Intel® oneAPI Math Kernel Library (oneMKL).

> **NOTE**
> These functions are available only for Intel® 64 architectures.

The cluster FFT function library was designed to perform fast Fourier transforms on a cluster, that is, a group of computers interconnected via a network. Each computer (node) in the cluster has its own memory and processor(s). Data interchanges between the nodes are provided by the network.

One or more processes may be running in parallel on each cluster node. To organize communication between different processes, the cluster FFT function library uses the Message Passing Interface (MPI). To avoid dependence on a specific MPI implementation (for example, MPICH, Intel® MPI, and others), the library works with MPI via a message-passing library for linear algebra called BLACS.

Cluster FFT functions of Intel® oneAPI Math Kernel Library (oneMKL) provide one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) functions and both Fortran and C interfaces for all transform functions.

To develop applications using the cluster FFT functions, you should have basic skills in MPI programming.

The interfaces for the Intel® oneAPI Math Kernel Library (oneMKL) cluster FFT functions are similar to the corresponding interfaces for the conventional Intel® oneAPI Math Kernel Library (oneMKL)FFT functions. Refer there for details not explained in this section.

Table "Cluster FFT Functions in Intel® oneAPI Math Kernel Library (oneMKL)" lists cluster FFT functions implemented in Intel® oneAPI Math Kernel Library (oneMKL):

Cluster FFT Functions in oneMKL

Function Name	Operation
Descriptor Manipulation Functions	
`DftiCreateDescriptorDM`	Allocates memory for the descriptor data structure and preliminarily initializes it.

Function Name	Operation
DftiCommitDescriptorDM	Performs all initialization for the actual FFT computation.
DftiFreeDescriptorDM	Frees memory allocated for a descriptor.
FFT Computation Functions	
DftiComputeForwardDM	Computes the forward FFT.
DftiComputeBackwardDM	Computes the backward FFT.
Descriptor Configuration Functions	
DftiSetValueDM	Sets one particular configuration parameter with the specified configuration value.
DftiGetValueDM	Gets the value of one particular configuration parameter.

Computing Cluster FFT

The Intel® oneAPI Math Kernel Library (oneMKL)cluster FFT functions are provided with Fortran and C interfaces. Fortran stands for Fortran 95.

Cluster FFT computation is performed by DftiComputeForwardDM and DftiComputeBackwardDM functions, called in a program using MPI, which will be referred to as MPI program. After an MPI program starts, a number of processes are created. MPI identifies each process by its rank. The processes are independent of one another and communicate via MPI. A function called in an MPI program is invoked in all the processes. Each process manipulates data according to its rank. Input or output data for a cluster FFT transform is a sequence of real or complex values. A cluster FFT computation function operates on the local part of the input data, that is, some part of the data to be operated in a particular process, as well as generates local part of the output data. While each process performs its part of computations, running in parallel and communicating through MPI, the processes perform the entire FFT computation. FFT computations using the Intel® oneAPI Math Kernel Library (oneMKL) cluster FFT functions are typically effected by a number of steps listed below:

1. Initiate MPI by calling `MPI_INIT` (the function must be called prior to calling any FFT function and any MPI function).
2. Allocate memory for the descriptor and create it by calling DftiCreateDescriptorDM.
3. Specify one of several values of configuration parameters by one or more calls to DftiSetValueDM.
4. Obtain values of configuration parameters needed to create local data arrays; the values are retrieved by calling DftiGetValueDM.
5. Initialize the descriptor for the FFT computation by calling DftiCommitDescriptorDM.
6. Create arrays for local parts of input and output data and fill the local part of input data with values. (For more information, see Distributing Data among Processes.)
7. Compute the transform by calling DftiComputeForwardDM or DftiComputeBackwardDM.
8. Gather local output data into the global array using MPI functions. (This step is optional because you may need to immediately employ the data differently.)
9. Release memory allocated for the descriptor by calling DftiFreeDescriptorDM.
10. Finalize communication through MPI by calling `MPI_FINALIZE` (the function must be called after the last call to a cluster FFT function and the last call to an MPI function).

Several code examples in Examples for Cluster FFT Functions in the Code Examples appendix illustrate cluster FFT computations.

Distributing Data Among Processes

The Intel® oneAPI Math Kernel Library (oneMKL) cluster FFT functions store all input and output multi-dimensional arrays (matrices) in one-dimensional arrays (vectors). The arrays are stored in the column-major order. For example, a two-dimensional matrix `A` of size (*m*,*n*) is stored in a vector `B` of size *m***n* so that

`B((j-1)*m+i)=A(i,j)` (*i=1, ..., m, j=1, ..., n*) .

NOTE
Order of FFT dimensions is the same as the order of array dimensions in the programming language. For example, a 3-dimensional FFT with Lengths=(*m,n,l*) can be computed over an array `AR(m,n,l)`.

All MPI processes involved in cluster FFT computation operate their own portions of data. These local arrays make up the virtual global array that the fast Fourier transform is applied to. It is your responsibility to properly allocate local arrays (if needed), fill them with initial data and gather resulting data into an actual global array or process the resulting data differently. To be able do this, see sections below on how the virtual global array is composed of the local ones.

Multi-dimensional transforms

If the dimension of transform is greater than one, the cluster FFT function library splits data in the dimension whose index changes most slowly, so that the parts contain all elements with several consecutive values of this index. It is the first dimension in C and the last dimension in Fortran. If the global array is two-dimensional, in C, it gives each process several consecutive rows. The term "rows" will be used regardless of the array dimension and programming language. Local arrays are placed in memory allocated for the virtual global array consecutively, in the order determined by process ranks. For example, in case of two processes, during the computation of a three-dimensional transform whose matrix has size (11,15,12), the processes may store local arrays of sizes (6,15,12) and (5,15,12), respectively.

If *p* is the number of MPI processes and the matrix of a transform to be computed has size (*m,n,l*), in C, each MPI process works with local data array of size (m_q , *n, l*), where $\Sigma m_q=m$, *q=0, ... , p-1*. Local input arrays must contain appropriate parts of the actual global input array, and then local output arrays will contain appropriate parts of the actual global output array. You can figure out which particular rows of the global array the local array must contain from the following configuration parameters of the cluster FFT interface: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, and `CDFT_LOCAL_SIZE`. To retrieve values of the parameters, use the `DftiGetValueDM` function:

- `CDFT_LOCAL_NX` specifies how many rows of the global array the current process receives.
- `CDFT_LOCAL_START_X` specifies which row of the global input or output array corresponds to the first row of the local input or output array. If `A` is a global array and `L` is the appropriate local array, then

  `L(i,j,k)=A(i,j,k+cdft_local_start_x-1)`, where *i=1, ..., m, j=1, ..., n, k=1, ..., l_q*.

Example "2D Out-of-place Cluster FFT Computation" shows how the data is distributed among processes for a two-dimensional cluster FFT computation.

One-dimensional transforms

In this case, input and output data are distributed among processes differently and even the numbers of elements stored in a particular process before and after the transform may be different. Each local array stores a segment of consecutive elements of the appropriate global array. Such segment is determined by the number of elements and a shift with respect to the first array element. So, to specify segments of the global input and output arrays that a particular process receives, *four* configuration parameters are needed: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, `CDFT_LOCAL_OUT_NX`, and `CDFT_LOCAL_OUT_START_X`. Use the `DftiGetValueDM` function to retrieve their values. The meaning of the four configuration parameters depends upon the type of the transform, as shown in Table "Data Distribution Configuration Parameters for 1D Transforms":

Data Distribution Configuration Parameters for 1D Transforms

Meaning of the Parameter	Forward Transform	Backward Transform
Number of elements in input array	`CDFT_LOCAL_NX`	`CDFT_LOCAL_OUT_NX`

Meaning of the Parameter	Forward Transform	Backward Transform
Elements shift in input array	CDFT_LOCAL_START_X	CDFT_LOCAL_OUT_START_X
Number of elements in output array	CDFT_LOCAL_OUT_NX	CDFT_LOCAL_NX
Elements shift in output array	CDFT_LOCAL_OUT_START_X	CDFT_LOCAL_START_X

Memory size for local data

The memory size needed for local arrays cannot be just calculated from CDFT_LOCAL_NX (CDFT_LOCAL_OUT_NX), because the cluster FFT functions sometimes require allocating a little bit more memory for local data than just the size of the appropriate sub-array. The configuration parameter CDFT_LOCAL_SIZE specifies the size of the local input and output array in data elements. Each local input and output arrays must have size not less than CDFT_LOCAL_SIZE**size_of_element*. Note that in the current implementation of the cluster FFT interface, data elements can be real or complex values, each complex value consisting of the real and imaginary parts. If you employ a user-defined workspace for in-place transforms (for more information, refer to Table "Settable configuration Parameters"), it must have the same size as the local arrays. Example "1D In-place Cluster FFT Computations" illustrates how the cluster FFT functions distribute data among processes in case of a one-dimensional FFT computation performed with a user-defined workspace.

Available Auxiliary Functions

If a global input array is located on one MPI process and you want to obtain its local parts or you want to gather the global output array on one MPI process, you can use functions MKL_CDFT_ScatterData and MKL_CDFT_GatherData to distribute or gather data among processes, respectively. These functions are defined in a file that is delivered with Intel® oneAPI Math Kernel Library (oneMKL) and located in the following subdirectory of the Intel® oneAPI Math Kernel Library (oneMKL) installation directory: examples/cdftf/source/cdft_example_support.f90.

Restriction on Lengths of Transforms

The algorithm that the Intel® oneAPI Math Kernel Library (oneMKL) cluster FFT functions use to distribute data among processes imposes a restriction on lengths of transforms with respect to the number of MPI processes used for the FFT computation:

- For a multi-dimensional transform, the lengths of the last two dimensions must be not less than the number of MPI processes.
- The length of a one-dimensional transform must be the product of two integers each of which is not less than the number of MPI processes.

Non-compliance with the restriction causes an error CDFT_SPREAD_ERROR (refer to Error Codes for details). To achieve the compliance, you can change the transform lengths and/or the number of MPI processes, which is specified at start of an MPI program. MPI-2 enables changing the number of processes during execution of an MPI program.

Cluster FFT Interface

To use the cluster FFT functions, you need to access the module MKL_CDFT through the "use" statement.

The Fortran interface provides a derived type DFTI_DESCRIPTOR_DM; a number of named constants representing various names of configuration parameters and their possible values; and a number of overloaded functions through the generic functionality of Fortran 95.

To provide communication between parallel processes through MPI, the following include statement must be present in your code:

- Fortran:

  ```
  INCLUDE "mpif.h"
  ```

  (for some MPI versions, `"mpif90.h"` header may be used instead).

There are three main categories of the cluster FFT functions in Intel® oneAPI Math Kernel Library (oneMKL):

1. **Descriptor Manipulation**. There are three functions in this category. The `DftiCreateDescriptorDM` function creates an FFT descriptor whose storage is allocated dynamically. The `DftiCommitDescriptorDM` function "commits" the descriptor to all its settings. The `DftiFreeDescriptorDM` function frees up the memory allocated for the descriptor.
2. **FFT Computation**. There are two functions in this category. The `DftiComputeForwardDM` function performs the forward FFT computation, and the `DftiComputeBackwardDM` function performs the backward FFT computation.
3. **Descriptor Configuration**. There are two functions in this category. The `DftiSetValueDM` function sets one specific configuration value to one of the many configuration parameters. The `DftiGetValueDM` function gets the current value of any of these configuration parameters, all of which are readable. These parameters, though many, are handled one at a time.

Cluster FFT Descriptor Manipulation Functions

There are three functions in this category: create a descriptor, commit a descriptor, and free a descriptor.

DftiCreateDescriptorDM

Allocates memory for the descriptor data structure and preliminarily initializes it.

Syntax

```
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, size)
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, sizes)
```

Include Files

- `mkl_cdft.f90`

Input Parameters

comm	MPI communicator, e.g. `MPI_COMM_WORLD`.
v1	Precision of the transform.
v2	Type of the forward domain. Must be `DFTI_COMPLEX` for complex-to-complex transforms or `DFTI_REAL` for real-to-complex transforms.
dim	Dimension of the transform.
size	Length of the transform in a one-dimensional case.
sizes	Lengths of the transform in a multi-dimensional case.

Output Parameters

handle	Pointer to the descriptor handle of transform. If the function completes successfully, the pointer to the created handle is stored in the variable.

Description

This function allocates memory in a particular MPI process for the descriptor data structure and instantiates it with default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. The result is a

pointer to the created descriptor. This function is slightly different from the "initialization" function DftiCommitDescriptorDM in a more traditional software packages or libraries used for computing the FFT. This function does not perform any significant computation work, such as twiddle factors computation, because the default configuration settings can still be changed using the function DftiSetValueDM.

The value of the parameter *v1* is specified through named constants DFTI_SINGLE and DFTI_DOUBLE. It corresponds to precision of input data, output data, and computation. A setting of DFTI_SINGLE indicates single-precision floating-point data type and a setting of DFTI_DOUBLE indicates double-precision floating-point data type.

The parameter *dim* is a simple positive integer indicating the dimension of the transform.

In Fortran, length is an integer or an array of integers.

Return Values

The function returns DFTI_NO_ERROR when completes successfully. In this case, the pointer to the created descriptor handle is stored in *handle*. If the function fails, it returns a value of another error class constant (for the list of constants, refer to Error Codes).

Interface

```
INTERFACE DftiCreateDescriptorDM
    INTEGER(4) FUNCTION DftiCreateDescriptorDMn(C,H,P1,P2,D,L)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: H
        INTEGER(4) C,P1,P2,D,L(*)
    END FUNCTION
    INTEGER(4) FUNCTION DftiCreateDescriptorDM1(C,H,P1,P2,D,L)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: H
        INTEGER(4) C,P1,P2,D,L
    END FUNCTION
END INTERFACE
```

DftiCommitDescriptorDM

Performs all initialization for the actual FFT computation.

Syntax

```
Status = DftiCommitDescriptorDM(handle)
```

Include Files

- mkl_cdft.f90

Input Parameters

handle	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM.

Description

The cluster FFT interface requires a function that completes initialization of a previously created descriptor before the descriptor can be used for FFT computations in a particular MPI process. The DftiCommitDescriptorDM function performs all initialization that facilitates the actual FFT computation. For the current implementation, it may involve exploring many different factorizations of the input length to search for a highly efficient computation method.

Any changes of configuration parameters of a committed descriptor via the set value function (see Descriptor Configuration Functions) requires a re-committal of the descriptor before a computation function can be invoked. Typically, this committal function is called right before a computation function call (see FFT Computation Functions).

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to Error Codes).

Interface

```
INTERFACE DftiCommitDescriptorDM
   INTEGER(4) FUNCTION DftiCommitDescriptorDM(handle);
     TYPE(DFTI_DESCRIPTOR_DM), POINTER :: handle
 END FUNCTION
END INTERFACE
```

DftiFreeDescriptorDM

Frees memory allocated for a descriptor.

Syntax

Status = DftiFreeDescriptorDM(*handle*)

Include Files

- mkl_cdft.f90

Input Parameters

handle	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM.

Output Parameters

handle	The descriptor handle. Memory allocated for the handle is released on output.

Description

This function frees up all memory allocated for a descriptor in a particular MPI process. Call the DftiFreeDescriptorDM function to delete the descriptor handle. Upon successful completion of DftiFreeDescriptorDM the descriptor handle is no longer valid.

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to Error Codes).

Interface

```
INTERFACE DftiFreeDescriptorDM
 INTEGER(4) FUNCTION DftiFreeDescriptorDM(handle)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: handle
```

```
   END FUNCTION
END INTERFACE
```

Cluster FFT Computation Functions

There are two functions in this category: compute the forward transform and compute the backward transform.

DftiComputeForwardDM

Computes the forward FFT.

Syntax

`Status = DftiComputeForwardDM(handle, in_X, out_X)`

`Status = DftiComputeForwardDM(handle, in_out_X)`

Include Files

- `mkl_cdft.f90`

Input Parameters

handle	The descriptor handle.
in_X, in_out_X	Local part of input data. Array of real or complex values (depending on the forward domain type). Refer to Distributing Data among Processes on how to allocate and initialize the array.

Output Parameters

out_X, in_out_X	Local part of output data. Array of complex values. Refer to Distributing Data among Processes on how to allocate the array.

Description

The `DftiComputeForwardDM` function computes the forward FFT. Forward FFT is the transform using the factor $e^{-i2\pi/n}$.

Before you call the function, the valid descriptor, created by `DftiCreateDescriptorDM`, must be configured and committed using the `DftiCommitDescriptorDM` function.

The computation is carried out by an internal call to the `DftiComputeForward` function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of `DftiComputeForward`.

The local part of input data, as well as the local part of the output data, is an appropriate sequence of real or complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See Distributing Data Among Processes for details.

Refer to Configuration Settings for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter `DFTI_PRECISION` determines the precision of input data, output data, and transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter DFTI_PLACEMENT informs the function whether the computation should be in-place. If the value of this parameter is DFTI_INPLACE (default), you must call the function with two parameters, otherwise you must supply three parameters. If DFTI_PLACEMENT = DFTI_INPLACE and three parameters are supplied, then the third parameter is ignored.

Caution
Even in case of an out-of-place transform, local array of input data *in_X* may be changed. To save data, make its copy before calling DftiComputeForwardDM.

In case of an in-place transform, DftiComputeForwardDM dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

NOTE
You can specify your own workspace of the same size through the configuration parameter CDFT_WORKSPACE to avoid redundant memory allocation.

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to Error Codes).

Interface

```
INTERFACE DftiComputeForwardDM
   INTEGER(4) FUNCTION DftiComputeForwardDM(h, in_X, out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
   COMPLEX(8), DIMENSION(*) :: in_x, out_X
   END FUNCTION DftiComputeForwardDM
   INTEGER(4) FUNCTION DftiComputeForwardDMi(h, in_out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(8), DIMENSION(*) :: in_out_X
   END FUNCTION DftiComputeForwardDMi
   INTEGER(4) FUNCTION DftiComputeForwardDMs(h, in_X, out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(4), DIMENSION(*) :: in_x, out_X
   END FUNCTION DftiComputeForwardDMs
   INTEGER(4) FUNCTION DftiComputeForwardDMis(h, in_out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(4), DIMENSION(*) :: in_out_X
   END FUNCTION DftiComputeForwardDMis
END INTERFACE
```

DftiComputeBackwardDM

Computes the backward FFT.

Syntax

```
Status = DftiComputeBackwardDM(handle, in_X, out_X)
```

```
Status = DftiComputeBackwardDM(handle, in_out_X)
```

Include Files

- mkl_cdft.f90

Input Parameters

handle	The descriptor handle.
in_X, in_out_X	Local part of input data. Array of complex values. Refer to Distributing Data among Processes on how to allocate and initialize the array.

Output Parameters

out_X, in_out_X	Local part of output data. Array of real or complex values (depending on the forward domain type. Refer to Distributing Data among Processes on how to allocate the array.

Description

The DftiComputeBackwardDM function computes the backward FFT. Backward FFT is the transform using the factor $e^{i2\pi/n}$.

Before you call the function, the valid descriptor, created by DftiCreateDescriptorDM, must be configured and committed using the DftiCommitDescriptorDM function.

The computation is carried out by an internal call to the DftiComputeBackward function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of DftiComputeBackward.

The local part of input data, as well as the local part of the output data, is an appropriate sequence of real or complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See Distributing Data among Processes for details.

Refer to Configuration Settings for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter DFTI_PRECISION determines the precision of input data, output data, and transform: a setting of DFTI_SINGLE indicates single-precision floating-point data type and a setting of DFTI_DOUBLE indicates double-precision floating-point data type.

The configuration parameter DFTI_PLACEMENT informs the function whether the computation should be in-place. If the value of this parameter is DFTI_INPLACE (default), you must call the function with two parameters, otherwise you must supply three parameters. If DFTI_PLACEMENT = DFTI_INPLACE and three parameters are supplied, then the third parameter is ignored.

Caution
Even in case of an out-of-place transform, local array of input data *in_X* may be changed. To save data, make its copy before calling DftiComputeBackwardDM.

In case of an in-place transform, DftiComputeBackwardDM dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

NOTE
You can specify your own workspace of the same size through the configuration parameter CDFT_WORKSPACE to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to Error Codes).

Interface

```
INTERFACE DftiComputeBackwardDM
   INTEGER(4) FUNCTION DftiComputeBackwardDM(h, in_X, out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(8), DIMENSION(*) :: in_x, out_X
   END FUNCTION DftiComputeBackwardDM
   INTEGER(4) FUNCTION DftiComputeBackwardDMi(h, in_out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(8), DIMENSION(*) :: in_out_X
   END FUNCTION DftiComputeBackwardDMi
   INTEGER(4) FUNCTION DftiComputeBackwardDMs(h, in_X, out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(4), DIMENSION(*) :: in_x, out_X
   END FUNCTION DftiComputeBackwardDMs
   INTEGER(4) FUNCTION DftiComputeBackwardDMis(h, in_out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(4), DIMENSION(*) :: in_out_X
   END FUNCTION DftiComputeBackwardDMis
END INTERFACE
```

Cluster FFT Descriptor Configuration Functions

There are two functions in this category: the value-setting function `DftiSetValueDM` sets one particular configuration parameter to an appropriate value, and the value-getting function `DftiGetValueDM` reads the value of one particular configuration parameter.

Some configuration parameters used by cluster FFT functions originate from the conventional FFT interface (see Configuration Settings for details).

Other configuration parameters are specific to the cluster FFT. Integer values of these parameters have type `INTEGER(4)`. The exact type of the configuration parameters being floating-point scalars is `REAL(4)` or `REAL(8)`. The configuration parameters whose values are named constants have the `INTEGER` type. They are defined in the `MKL_CDFT` module.

The names of the configuration parameters specific to the cluster FFT interface have the `CDFT` prefix.

DftiSetValueDM

Sets one particular configuration parameter with the specified configuration value.

Syntax

`Status = DftiSetValueDM (`*`handle`*`, `*`param`*`, `*`value`*`)`

Include Files

- `mkl_cdft.f90`

Input Parameters

`handle`	The descriptor handle. Must be valid, that is, created in a call to `DftiCreateDescriptorDM`.

param Name of a parameter to be set up in the descriptor handle. See Table "Settable Configuration Parameters" for the list of available parameters.

value Value of the parameter.

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in the table below, and the configuration value must have the corresponding type. See Configuration Settings for details of the meaning of each setting and for possible values of the parameters whose values are named constants.

Settable Configuration Parameters

Parameter Name	Data Type	Description	Default Value
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor of forward transform.	1.0
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor of backward transform.	1.0
DFTI_PLACEMENT	Named constant	Placement of the computation result.	DFTI_INPLACE
DFTI_ORDERING	Named constant	Scrambling of data order.	DFTI_ORDERED
CDFT_WORKSPACE	Array of an appropriate type	Auxiliary buffer, a user-defined workspace. Enables saving memory during in-place computations.	NULL (allocate workspace dynamically).
DFTI_PACKED_FORMAT	Named constant	Packed format for storing conjugate-even sequence (in the case of a real forward domain).	• DFTI_PERM_FORMAT — default and the only available value for one-dimensional transforms • DFTI_CCE_FORMAT — default and the only available value for multi-dimensional transforms
DFTI_TRANSPOSE	Named constant	This parameter determines how the output data is located for multi-dimensional transforms. If the parameter value is DFTI_NONE, the data is located in a usual manner described in this document. If the value is DFTI_ALLOW, the last (first) global transposition is not performed for a forward (backward) transform.	DFTI_NONE

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to Error Codes).

Interface

```
INTERFACE DftiSetValueDM
   INTEGER(4) FUNCTION DftiSetValueDM(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p, v
   END FUNCTION
   INTEGER(4) FUNCTION DftiSetValueDMd(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      REAL(8) :: v
   END FUNCTION
   INTEGER(4) FUNCTION DftiSetValueDMs(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      REAL(4) :: v
   END FUNCTION
   INTEGER(4) FUNCTION DftiSetValueDMsw(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      COMPLEX(4) :: v(*)
   END FUNCTION
   INTEGER(4) FUNCTION DftiSetValueDMdw(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      COMPLEX(8) :: v(*)
   END FUNCTION
END INTERFACE
```

DftiGetValueDM

Gets the value of one particular configuration parameter.

Syntax

`Status = DftiGetValueDM(`*`handle`*`, `*`param`*`, `*`value`*`)`

Include Files

- `mkl_cdft.f90`

Input Parameters

`handle`	The descriptor handle. Must be valid, that is, created in a call to `DftiCreateDescriptorDM`.
`param`	Name of a parameter to be retrieved from the descriptor. See Table "Retrievable Configuration Parameters" for the list of available parameters.

Output Parameters

value	Value of the parameter.

Description

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in the table below, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type. Possible values of the named constants can be found in Table "Configuration Parameters" and relevant subsections of the Configuration Settings section.

Retrievable Configuration Parameters

Parameter Name	Data Type	Description
DFTI_PRECISION	Named constant	Precision of computation, input data and output data.
DFTI_DIMENSION	Integer scalar	Dimension of the transform
DFTI_LENGTHS	Array of integer values	Array of lengths of the transform. Number of lengths corresponds to the dimension of the transform.
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor of forward transform.
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor of backward transform.
DFTI_PLACEMENT	Named constant	Placement of the computation result.
DFTI_COMMIT_STATUS	Named constant	Shows whether descriptor has been committed.
DFTI_FORWARD_DOMAIN	Named constant	Forward domain of transforms, has the value of DFTI_COMPLEX or DFTI_REAL.
DFTI_ORDERING	Named constant	Scrambling of data order.
CDFT_MPI_COMM	Type of MPI communicator	MPI communicator used for transforms.
CDFT_LOCAL_SIZE	Integer scalar	Necessary size of input, output, and buffer arrays in data elements.
CDFT_LOCAL_X_START	Integer scalar	Row/element number of the global array that corresponds to the first row/element of the local array. For more information, see Distributing Data among Processes.
CDFT_LOCAL_NX	Integer scalar	The number of rows/elements of the global array stored in the local array. For more information, see Distributing Data among Processes.
CDFT_LOCAL_OUT_X_START	Integer scalar	Element number of the appropriate global array that corresponds to the first element of the input or output local array in a 1D case. For details, see Distributing Data among Processes.

Parameter Name	Data Type	Description
CDFT_LOCAL_OUT_NX	Integer scalar	The number of elements of the appropriate global array that are stored in the input or output local array in a 1D case. For details, see Distributing Data among Processes.

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to Error Codes).

Interface

```
INTERFACE DftiGetValueDM
   INTEGER(4) FUNCTION DftiGetValueDM(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p, v
   END FUNCTION
   INTEGER(4) FUNCTION DftiGetValueDMar(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p, v(*)
   END FUNCTION
   INTEGER(4) FUNCTION DftiGetValueDMd(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      REAL(8) :: v
   END FUNCTION
   INTEGER(4) FUNCTION DftiGetValueDMs(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      REAL(4) :: v
   END FUNCTION
END INTERFACE
```

Error Codes

All the cluster FFT functions return an integer value denoting the status of the operation. These values are identified by named constants. Each function returns DFTI_NO_ERROR if no errors were encountered during execution. Otherwise, a function generates an error code. In addition to FFT error codes, the cluster FFT interface has its own ones. The named constants specific to the cluster FFT interface have the CDFT prefix in their names. Table "Error Codes that Cluster FFT Functions Return" lists error codes that the cluster FFT functions may return.

Error Codes that Cluster FFT Functions Return

Named Constants	Comments
DFTI_NO_ERROR	No error.
DFTI_MEMORY_ERROR	Usually associated with memory allocation.
DFTI_INVALID_CONFIGURATION	Invalid settings of one or more configuration parameters.
DFTI_INCONSISTENT_CONFIGURATION	Inconsistent configuration or input parameters.

Named Constants	Comments
DFTI_NUMBER_OF_THREADS_ERROR	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function).
DFTI_MULTITHREADED_ERROR	Usually associated with a value that OMP routines return in case of errors.
DFTI_BAD_DESCRIPTOR	Descriptor is unusable for computation.
DFTI_UNIMPLEMENTED	Unimplemented legitimate settings; implementation dependent.
DFTI_MKL_INTERNAL_ERROR	Internal library error.
DFTI_1D_LENGTH_EXCEEDS_INT32	Length of one of the dimensions exceeds 2^{32} -1 (4 bytes).
CDFT_SPREAD_ERROR	Data cannot be distributed (For more information, see Distributing Data among Processes.)
CDFT_MPI_ERROR	MPI error. Occurs when calling MPI.

PBLAS Routines

Intel® oneAPI Math Kernel Libraryimplements the PBLAS (Parallel Basic Linear Algebra Subprograms) routines from the ScaLAPACK package for distributed-memory architecture. PBLAS is intended for using in vector-vector, matrix-vector, and matrix-matrix operations to simplify the parallelization of linear codes. The design of PBLAS is as consistent as possible with that of the BLAS. The routine descriptions are arranged in several sections according to the PBLAS level of operation:

- PBLAS Level 1 Routines (distributed vector-vector operations)
- PBLAS Level 2 Routines (distributed matrix-vector operations)
- PBLAS Level 3 Routines (distributed matrix-matrix operations)

Each section presents the routine and function group descriptions in alphabetical order by the routine group name; for example, the `p?asum` group, the `p?axpy` group. The question mark in the group name corresponds to a character indicating the data type (`s`, `d`, `c`, and `z` or their combination); see *Routine Naming Conventions*.

NOTE
PBLAS routines are provided only with Intel® oneAPI Math Kernel Library (oneMKL) versions for Linux* and Windows* OSs.

Generally, PBLAS runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of PBLAS optimized for the target architecture. The Intel® oneAPI Math Kernel Library (oneMKL) version of PBLAS is optimized for Intel® processors. For the detailed system and environment requirements see*Intel® oneAPI Math Kernel Library (oneMKL) Release Notes* and *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

For full reference on PBLAS routines and related information, see http://www.netlib.org/scalapack/html/pblas_qref.html.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

PBLAS Routines Overview

The model of the computing environment for PBLAS is represented as a one-dimensional array of processes or also a two-dimensional process grid. To use PBLAS, all global matrices or vectors must be distributed on this array or grid prior to calling the PBLAS routines.

PBLAS uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, as well as gives the opportunity to use PBLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global array and its corresponding process and memory location is contained in the so called *array descriptor* associated with each global array. Table "Content of the array descriptor for dense matrices" gives an example of an array descriptor structure.

Content of Array Descriptor for Dense Matrices

Array Element #	Name	Definition
1	*dtype*	Descriptor type (=1 for dense matrices)
2	*ctxt*	BLACS context handle for the process grid
3	*m*	Number of rows in the global array
4	*n*	Number of columns in the global array
5	*mb*	Row blocking factor
6	*nb*	Column blocking factor
7	*rsrc*	Process row over which the first row of the global array is distributed
8	*csrc*	Process column over which the first column of the global array is distributed
9	*lld*	Leading dimension of the local array

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by *LOCr*() and *LOCc*(), respectively. To compute these numbers, you can use the ScaLAPACK tool routine numroc.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix of the global matrix *A*, which is contained in the global subarray sub(*A*), defined by the following 6 values (for dense matrices):

m	The number of rows of sub(*A*)
n	The number of columns of sub(*A*)
a	A pointer to the local array containing the entire global array *A*
ia	The row index of sub(*A*) in the global array
ja	The column index of sub(*A*) in the global array
desca	The array descriptor for the global array *A*

Intel® oneAPI Math Kernel Library (oneMKL) provides the PBLAS routines with interface similar to the interface used in the Netlib PBLAS (see http://www.netlib.org/scalapack/html/pblas_qref.html).

PBLAS Routine Naming Conventions

The naming convention for PBLAS routines is similar to that used for BLAS routines (see Routine Naming Conventions). A general rule is that each routine name in PBLAS, which has a BLAS equivalent, is simply the BLAS name prefixed by initial letter p that stands for "parallel".

The Intel® oneAPI Math Kernel Library (oneMKL) PBLAS routine names have the following structure:

```
p <character> <name> <mod> ( )
```

The <*character*> field indicates the Fortran data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision
i	integer

Some routines and functions can have combined character codes, such as sc or dz.

For example, the function pscasum uses a complex input array and returns a real value.

The <*name*> field, in PBLAS level 1, indicates the operation type. For example, the PBLAS level 1 routines p?dot, p?swap, p?copy compute a vector dot product, vector swap, and a copy vector, respectively.

In PBLAS level 2 and 3, <*name*> reflects the matrix argument type:

ge	general matrix
sy	symmetric matrix
he	Hermitian matrix
tr	triangular matrix

In PBLAS level 3, the <*name*>=tran indicates the transposition of the matrix.

The <*mod*> field, if present, provides additional details of the operation. The PBLAS level 1 names can have the following characters in the <*mod*> field:

c	conjugated vector
u	unconjugated vector

The PBLAS level 2 names can have the following additional characters in the <*mod*> field:

mv	matrix-vector product
sv	solving a system of linear equations with matrix-vector operations
r	rank-1 update of a matrix
r2	rank-2 update of a matrix.

The PBLAS level 3 names can have the following additional characters in the <*mod*> field:

mm	matrix-matrix product
sm	solving a system of linear equations with matrix-matrix operations
rk	rank-*k* update of a matrix
r2k	rank-2*k* update of a matrix.

The examples below show how to interpret PBLAS routine names:

pddot	<p> <d> <dot>: double-precision real distributed vector-vector dot product
pcdotc	<p> <c> <dot> <c>: complex distributed vector-vector dot product, conjugated
pscasum	<p> <sc> <asum>: sum of magnitudes of distributed vector elements, single precision real output and single precision complex input
pcdotu	<p> <c> <dot> <u>: distributed vector-vector dot product, unconjugated, complex

psgemv	<p> <s> <ge> <mv>: distributed matrix-vector product, general matrix, single precision
pztrmm	<p> <z> <tr> <mm>: distributed matrix-matrix product, triangular matrix, double-precision complex.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

PBLAS Level 1 Routines

PBLAS Level 1 includes routines and functions that perform distributed vector-vector operations. Table "PBLAS Level 1 Routine Groups and Their Data Types" lists the PBLAS Level 1 routine groups and the data types associated with them.

PBLAS Level 1 Routine Groups and Their Data Types

Routine or Function Group	Data Types	Description
p?amax	s, d, c, z	Calculates an index of the distributed vector element with maximum absolute value
p?asum	s, d, sc, dz	Calculates sum of magnitudes of a distributed vector
p?axpy	s, d, c, z	Calculates distributed vector-scalar product
p?copy	s, d, c, z	Copies a distributed vector
p?dot	s, d	Calculates a dot product of two distributed real vectors
p?dotc	c, z	Calculates a dot product of two distributed complex vectors, one of them is conjugated
p?dotu	c, z	Calculates a dot product of two distributed complex vectors
p?nrm2	s, d, sc, dz	Calculates the 2-norm (Euclidean norm) of a distributed vector
p?scal	s, d, c, z, cs, zd	Calculates a product of a distributed vector by a scalar
p?swap	s, d, c, z	Swaps two distributed vectors

p?amax

Computes the global index of the element of a distributed vector with maximum absolute value.

Syntax

```
call psamax(n, amax, indx, x, ix, jx, descx, incx)
call pdamax(n, amax, indx, x, ix, jx, descx, incx)
call pcamax(n, amax, indx, x, ix, jx, descx, incx)
call pzamax(n, amax, indx, x, ix, jx, descx, incx)
```

Include Files

- mkl_pblas.h

Description

The functions p?amax compute global index of the maximum element in absolute value of a distributed vector sub(*x*),

where sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx*=*m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx*= 1.

Input Parameters

n	(global) INTEGER. The length of distributed vector sub(*x*), *n*≥0.
x	(local) REAL for psamax DOUBLE PRECISION for pdamax COMPLEX for pcamax DOUBLE COMPLEX for pzamax Array, size (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)). This array contains the entries of the distributed vector sub(*x*).
ix, *jx*	(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.
descx	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.
incx	(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

amax	(global)REAL for psamax. DOUBLE PRECISION for pdamax. COMPLEX for pcamax. DOUBLE COMPLEX for pzamax. Maximum absolute value (magnitude) of elements of the distributed vector only in its scope.
indx	(global) INTEGER. The global index of the maximum element in absolute value of the distributed vector sub(*x*) only in its scope.

p?asum

Computes the sum of magnitudes of elements of a distributed vector.

Syntax

```
call psasum(n, asum, x, ix, jx, descx, incx)
call pscasum(n, asum, x, ix, jx, descx, incx)
```

```
call pdasum(n, asum, x, ix, jx, descx, incx)
call pdzasum(n, asum, x, ix, jx, descx, incx)
```

Include Files

- mkl_pblas.h

Description

The functions p?asum compute the sum of the magnitudes of elements of a distributed vector sub(*x*), where sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx*=*m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx*= 1.

Input Parameters

n	(global) INTEGER. The length of distributed vector sub(*x*), *n*≥0.
x	(local) REAL for psasum DOUBLE PRECISION for pdasum COMPLEX for pscasum DOUBLE COMPLEX for pdzasum Array, size (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)). This array contains the entries of the distributed vector sub(*x*).
ix, *jx*	(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.
descx	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.
incx	(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

asum	(local) REAL for psasum and pscasum. DOUBLE PRECISION for pdasum and pdzasum Contains the sum of magnitudes of elements of the distributed vector only in its scope.

p?axpy

Computes a distributed vector-scalar product and adds the result to a distributed vector.

Syntax

```
call psaxpy(n, a, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pdaxpy(n, a, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pcaxpy(n, a, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzaxpy(n, a, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?axpy routines perform the following operation with distributed vectors:

```
sub(y) := sub(y) + a*sub(x)
```

where:

a is a scalar;

sub(*x*) and sub(*y*) are *n*-element distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx*=*m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx*= 1;

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy*=*m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy*= 1.

Input Parameters

n	(global) INTEGER. The length of distributed vectors, *n*≥0.
a	(local) REAL for psaxpy DOUBLE PRECISION for pdaxpy COMPLEX for pcaxpy DOUBLE COMPLEX for pzaxpy Specifies the scalar *a*.
x	(local) REAL for psaxpy DOUBLE PRECISION for pdaxpy COMPLEX for pcaxpy DOUBLE COMPLEX for pzaxpy Array, size (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)). This array contains the entries of the distributed vector sub(*x*).
ix, *jx*	(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.
descx	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.
incx	(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.
y	(local) REAL for psaxpy DOUBLE PRECISION for pdaxpy COMPLEX for pcaxpy DOUBLE COMPLEX for pzaxpy Array, size (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)). This array contains the entries of the distributed vector sub(*y*).

iy, *jy*	(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.
descy	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.
incy	(global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

y	Overwritten by sub(*y*) := sub(*y*) + *a**sub(*x*).

p?copy

Copies one distributed vector to another vector.

Syntax

```
call pscopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pdcopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pccopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzcopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call picopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?copy routines perform a copy operation with distributed vectors defined as

```
sub(y) = sub(x),
```

where sub(*x*) and sub(*y*) are *n*-element distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx*=*m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx*= 1;

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy*=*m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy*= 1.

Input Parameters

n	(global) INTEGER. The length of distributed vectors, *n*≥0.
x	(local) REAL for pscopy DOUBLE PRECISION for pdcopy COMPLEX for pccopy DOUBLE COMPLEX for pzcopy INTEGER for picopy Array, size (*jx*-1)**m_x* + *ix*+(*n*-1)*abs(*incx*)). This array contains the entries of the distributed vector sub(*x*).

ix, *jx*	(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.
descx	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.
incx	(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.
y	(local) REAL for pscopy DOUBLE PRECISION for pdcopy COMPLEX for pccopy DOUBLE COMPLEX for pzcopy INTEGER for picopy Array, size (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)). This array contains the entries of the distributed vector sub(*y*).
iy, *jy*	(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.
descy	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.
incy	(global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

y	Overwritten with the distributed vector sub(*x*).

p?dot

Computes the dot product of two distributed real vectors.

Syntax

```
call psdot(n, dot, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pddot(n, dot, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The ?dot functions compute the dot product *dot* of two distributed real vectors defined as

```
dot  = sub(x)'*sub(y)
```

where sub(*x*) and sub(*y*) are *n*-element distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx*=*m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx*= 1;

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy*=*m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy*= 1.

Input Parameters

n — (global) INTEGER. The length of distributed vectors, *n*≥0.

x — (local) REAL for psdot

DOUBLE PRECISION for pddot

Array, size (*jx*-1)**m_x* + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx* — (global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx — (global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

y — (local)REAL for psdot

DOUBLE PRECISION for pddot

Array, size (*jy*-1)**m_y* + *iy*+(*n*-1)*abs(*incy*)).

This array contains the entries of the distributed vector sub(*y*).

iy, *jy* — (global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.

descy — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy — (global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

dot — (local) REAL for psdot

DOUBLE PRECISION for pddot

Dot product of sub(*x*) and sub(*y*) only in their scope.

p?dotc

Computes the dot product of two distributed complex vectors, one of them is conjugated.

Syntax

```
call pcdotc(n, dotc, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzdotc(n, dotc, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?dotc functions compute the dot product *dotc* of two distributed vectors, with one vector conjugated:

```
dotc  = conjg(sub(x)')*sub(y)
```

where sub(*x*) and sub(*y*) are *n*-element distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx*=*m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx*= 1;

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy*=*m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy*= 1.

Input Parameters

n
(global) INTEGER. The length of distributed vectors, *n*≥0.

x
(local)COMPLEX for pcdotc

DOUBLE COMPLEX for pzdotc

Array, size (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*
(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx
(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx
(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

y
(local) COMPLEX for pcdotc

DOUBLE COMPLEX for pzdotc

Array, size (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)).

This array contains the entries of the distributed vector sub(*y*).

iy, *jy*
(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.

descy
(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy
(global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

dotc
(local) COMPLEX for pcdotc

DOUBLE COMPLEX for pzdotc

Dot product of sub(*x*) and sub(*y*) only in their scope.

p?dotu

Computes the dot product of two distributed complex vectors.

Syntax

```
call pcdotu(n, dotu, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzdotu(n, dotu, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?dotu functions compute the dot product *dotu* of two distributed vectors defined as

```
dotu  = sub(x)'*sub(y)
```

where sub(*x*) and sub(*y*) are *n*-element distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx*=*m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx*= 1;

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy*=*m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy*= 1.

Input Parameters

n
(global) INTEGER. The length of distributed vectors, *n*≥0.

x
(local) COMPLEX for pcdotu
DOUBLE COMPLEX for pzdotu
Array, size (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).
This array contains the entries of the distributed vector sub(*x*).

ix, *jx*
(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx
(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx
(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

y
(local) COMPLEX for pcdotu
DOUBLE COMPLEX for pzdotu
Array, size (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)).
This array contains the entries of the distributed vector sub(*y*).

iy, *jy*
(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.

descy
(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy (global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

dotu (local) COMPLEX for pcdotu

DOUBLE COMPLEX for pzdotu

Dot product of sub(*x*) and sub(*y*) only in their scope.

p?nrm2

Computes the Euclidean norm of a distributed vector.

Syntax

```
call psnrm2(n, norm2, x, ix, jx, descx, incx)
call pdnrm2(n, norm2, x, ix, jx, descx, incx)
call pscnrm2(n, norm2, x, ix, jx, descx, incx)
call pdznrm2(n, norm2, x, ix, jx, descx, incx)
```

Include Files

- mkl_pblas.h

Description

The p?nrm2 functions compute the Euclidean norm of a distributed vector sub(*x*),

where sub(*x*) is an *n*-element distributed vector.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx*=*m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx*= 1.

Input Parameters

n (global) INTEGER. The length of distributed vector sub(*x*), *n*≥0.

x (local) REAL for psnrm2

DOUBLE PRECISION for pdnrm2

COMPLEX for pscnrm2

DOUBLE COMPLEX for pdznrm2

Array, size (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx* (global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx (global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

norm2 (local) REAL for psnrm2 and pscnrm2.

DOUBLE PRECISION for pdnrm2 and pdznrm2

Contains the Euclidean norm of a distributed vector only in its scope.

p?scal

Computes a product of a distributed vector by a scalar.

Syntax

```
call psscal(n, a, x, ix, jx, descx, incx)
call pdscal(n, a, x, ix, jx, descx, incx)
call pcscal(n, a, x, ix, jx, descx, incx)
call pzscal(n, a, x, ix, jx, descx, incx)
call pcsscal(n, a, x, ix, jx, descx, incx)
call pzdscal(n, a, x, ix, jx, descx, incx)
```

Include Files

- mkl_pblas.h

Description

The p?scal routines multiplies a n-element distributed vector sub(x) by the scalar a:

```
sub(x) = a*sub(x),
```

where sub(x) denotes $X(ix, jx:jx+n-1)$ if $incx=m\_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$.

Input Parameters

n (global) INTEGER. The length of distributed vector sub(x), $n \geq 0$.

a (global) REAL for psscal and pcsscal

DOUBLE PRECISION for pdscal and pzdscal

COMPLEX for pcscal

DOUBLE COMPLEX for pzscal

Specifies the scalar a.

x (local) REAL for psscal

DOUBLE PRECISION for pdscal

COMPLEX for pcscal and pcsscal

DOUBLE COMPLEX for pzscal and pzdscal

Array, size $(jx-1)*m\_x + ix+(n-1)*abs(incx))$.

This array contains the entries of the distributed vector sub(x).

ix, *jx*	(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.
descx	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.
incx	(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

x	Overwritten by the updated distributed vector sub(*x*)

p?swap

Swaps two distributed vectors.

Syntax

```
call psswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pdswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pcswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

Given two distributed vectors sub(*x*) and sub(*y*), the p?swap routines return vectors sub(*y*) and sub(*x*) swapped, each replacing the other.

Here sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx*=*m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx*= 1;

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy*=*m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy*= 1.

Input Parameters

n	(global) INTEGER. The length of distributed vectors, *n*≥0.
x	(local) REAL for psswap DOUBLE PRECISION for pdswap COMPLEX for pcswap DOUBLE COMPLEX for pzswap Array, size (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)). This array contains the entries of the distributed vector sub(*x*).
ix, *jx*	(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.
incx	(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.
y	(local) REAL for psswap DOUBLE PRECISION for pdswap COMPLEX for pcswap DOUBLE COMPLEX for pzswap Array, size (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)). This array contains the entries of the distributed vector sub(*y*).
iy, *jy*	(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.
descy	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.
incy	(global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

x	Overwritten by distributed vector sub(*y*).
y	Overwritten by distributed vector sub(*x*).

PBLAS Level 2 Routines

This section describes PBLAS Level 2 routines, which perform distributed matrix-vector operations. Table "PBLAS Level 2 Routine Groups and Their Data Types" lists the PBLAS Level 2 routine groups and the data types associated with them.

PBLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
p?gemv	s, d, c, z	Matrix-vector product using a distributed general matrix
p?agemv	s, d, c, z	Matrix-vector product using absolute values for a distributed general matrix
p?ger	s, d	Rank-1 update of a distributed general matrix
p?gerc	c, z	Rank-1 update (conjugated) of a distributed general matrix
p?geru	c, z	Rank-1 update (unconjugated) of a distributed general matrix
p?hemv	c, z	Matrix-vector product using a distributed Hermitian matrix
p?ahemv	c, z	Matrix-vector product using absolute values for a distributed Hermitian matrix

Routine Groups	Data Types	Description
p?her	c, z	Rank-1 update of a distributed Hermitian matrix
p?her2	c, z	Rank-2 update of a distributed Hermitian matrix
p?symv	s, d	Matrix-vector product using a distributed symmetric matrix
p?asymv	s, d	Matrix-vector product using absolute values for a distributed symmetric matrix
p?syr	s, d	Rank-1 update of a distributed symmetric matrix
p?syr2	s, d	Rank-2 update of a distributed symmetric matrix
p?trmv	s, d, c, z	Distributed matrix-vector product using a triangular matrix
p?atrmv	s, d, c, z	Distributed matrix-vector product using absolute values for a triangular matrix
p?trsv	s, d, c, z	Solves a system of linear equations whose coefficients are in a distributed triangular matrix

p?gemv

Computes a distributed matrix-vector product using a general matrix.

Syntax

```
call psgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
call pdgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
call pcgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
call pzgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?gemv routines perform a distributed matrix-vector operation defined as

```
sub(y)  := alpha*sub(A)*sub(x) + beta*sub(y),
```

or

```
sub(y)  := alpha*sub(A)'*sub(x) + beta*sub(y),
```

or

```
sub(y)  := alpha*conjg(sub(A)')*sub(x) + beta*sub(y),
```

where

alpha and *beta* are scalars,

sub(*A*) is a *m*-by-*n* submatrix, sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1),

sub(*x*) and sub(*y*) are subvectors.

When *trans* = 'N' or 'n', sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1, sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*m*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*m*-1, *jy*) if *incy* = 1.

When *trans* = 'T' or 't', or 'C', or 'c', sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*m*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*m*-1, *jx*) if *incx* = 1, sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*m*-1, *jy*) if *incy* = 1.

Input Parameters

trans

(global) CHARACTER*1. Specifies the operation:

if *trans*= 'N' or 'n', then sub(*y*) := *alpha**sub(*A*)'*sub(*x*) + *beta**sub(*y*);

if *trans*= 'T' or 't', then sub(*y*) := *alpha**sub(*A*)'*sub(*x*) + *beta**sub(*y*);

if *trans*= 'C' or 'c', then sub(*y*) := *alpha**conjg(sub*A*)')*sub(*x*) + *beta**sub(*y*).

m

(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*A*), *m*≥0.

n

(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*A*), *n*≥0.

alpha

(global)REAL for psgemv

DOUBLE PRECISION for pdgemv

COMPLEX for pcgemv

DOUBLE COMPLEX for pzgemv

Specifies the scalar *alpha*.

a

(local)REAL for psgemv

DOUBLE PRECISION for pdgemv

COMPLEX for pcgemv

DOUBLE COMPLEX for pzgemv

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). Before entry this array must contain the local pieces of the distributed matrix sub(*A*).

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x

(local)REAL for psgemv

DOUBLE PRECISION for pdgemv

COMPLEX for pcgemv

DOUBLE COMPLEX for pzgemv

Array, size (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)) when *trans* = 'N' or 'n', and (*jx*-1)*m_x + *ix*+(*m*-1)*abs(*incx*)) otherwise.

This array contains the entries of the distributed vector sub(*x*).

ix, *jx* — (global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx — (global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

beta — (global)REAL for psgemv

DOUBLE PRECISION for pdgemv

COMPLEX for pcgemv

DOUBLE COMPLEX for pzgemv

Specifies the scalar *beta*. When *beta* is set to zero, then sub(*y*) need not be set on input.

y — (local)REAL for psgemv

DOUBLE PRECISION for pdgemv

COMPLEX for pcgemv

DOUBLE COMPLEX for pzgemv

Array, size (*jy*-1)*m_y + *iy*+(*m*-1)*abs(*incy*)) when *trans* = 'N' or 'n', and (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)) otherwise.

This array contains the entries of the distributed vector sub(*y*).

iy, *jy* — (global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.

descy — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy — (global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

y — Overwritten by the updated distributed vector sub(*y*).

p?agemv

Computes a distributed matrix-vector product using absolute values for a general matrix.

Syntax

call psagemv(*trans*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

call pdagemv(*trans*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

call pcagemv(*trans*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

call pzagemv(*trans*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

Include Files

- mkl_pblas.h

Description

The p?agemv routines perform a distributed matrix-vector operation defined as

```
sub(y)  := abs(alpha)*abs(sub(A)')*abs(sub(x)) + abs(beta*sub(y)),
```

or

```
sub(y)  := abs(alpha)*abs(sub(A)')*abs(sub(x)) + abs(beta*sub(y)),
```

or

```
sub(y)  := abs(alpha)*abs(conjg(sub(A)'))*abs(sub(x)) + abs(beta*sub(y)),
```

where

alpha and *beta* are scalars,

sub(*A*) is a *m*-by-*n* submatrix, sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1),

sub(*x*) and sub(*y*) are subvectors.

When *trans* = 'N' or 'n',

sub(*x*) denotes *X*(*ix*:*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and

X(*ix*:*ix*+*n*-1, *jx*:*jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*:*iy*, *jy*:*jy*+*m*-1) if *incy* = *m_y*, and

Y(*iy*:*iy*+*m*-1, *jy*:*jy*) if *incy* = 1.

When *trans* = 'T' or 't', or 'C', or 'c',

sub(*x*) denotes *X*(*ix*:*ix*, *jx*:*jx*+*m*-1) if *incx* = *m_x*, and

X(*ix*:*ix*+*m*-1, *jx*:*jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*:*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and

Y(*iy*:*iy*+*m*-1, *jy*:*jy*) if *incy* = 1.

Input Parameters

trans (global) CHARACTER*1. Specifies the operation:

if *trans*= 'N' or 'n', then sub(*y*) := |*alpha*|*|sub(*A*)|*|sub(*x*)| + |*beta**sub(*y*)|

if *trans*= 'T' or 't', then sub(*y*) := |*alpha*|*|sub(*A*)'|*|sub(*x*)| + |*beta**sub(*y*)|

if *trans*= 'C' or 'c', then sub(*y*) := |*alpha*|*|sub(*A*)'|*|sub(*x*)| + |*beta**sub(*y*)|.

m (global) INTEGER. Specifies the number of rows of the distributed matrix sub(*A*), *m*≥0.

n (global) INTEGER. Specifies the number of columns of the distributed matrix sub(*A*), *n*≥0.

alpha (global)REAL for psagemv

DOUBLE PRECISION for pdagemv

COMPLEX for pcagemv

DOUBLE COMPLEX for pzagemv

Specifies the scalar *alpha*.

a (local)REAL for psagemv

DOUBLE PRECISION for pdagemv

COMPLEX for pcagemv

DOUBLE COMPLEX for pzagemv

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). Before entry this array must contain the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x (local)REAL for psagemv

DOUBLE PRECISION for pdagemv

COMPLEX for pcagemv

DOUBLE COMPLEX for pzagemv

Array, size (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)) when *trans* = 'N' or 'n', and (*jx*-1)*m_x + *ix*+(*m*-1)*abs(*incx*)) otherwise.

This array contains the entries of the distributed vector sub(*x*).

ix, *jx* (global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx (global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

beta	(global)REAL for psagemv DOUBLE PRECISION for pdagemv COMPLEX for pcagemv DOUBLE COMPLEX for pzagemv Specifies the scalar *beta*. When *beta* is set to zero, then sub(*y*) need not be set on input.
y	(local)REAL for psagemv DOUBLE PRECISION for pdagemv COMPLEX for pcagemv DOUBLE COMPLEX for pzagemv Array, size (*jy*-1)*m_y + *iy*+(*m*-1)*abs(*incy*)) when *trans* = 'N' or 'n', and (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)) otherwise. This array contains the entries of the distributed vector sub(*y*).
iy, *jy*	(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.
descy	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.
incy	(global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

y	Overwritten by the updated distributed vector sub(*y*).

p?ger

Performs a rank-1 update of a distributed general matrix.

Syntax

```
call psger(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
call pdger(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

Include Files

- mkl_pblas.h

Description

The p?ger routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*sub(y)' + sub(A),
```

where:

alpha is a scalar,

sub(*A*) is a *m*-by-*n* distributed general matrix, sub(*A*)=*A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1),

sub(*x*) is an *m*-element distributed vector, sub(*y*) is an *n*-element distributed vector,

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*m*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*m*-1, *jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

m
(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*A*), *m*≥0.

n
(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*A*), *n*≥0.

alpha
(global)REAL for psger

DOUBLE REAL for pdger

Specifies the scalar *alpha*.

x
(local)REAL for psger

DOUBLE REAL for pdger

Array, size at least (*jx*-1)*m_x + *ix*+(*m*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*
(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx
(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx
(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

y
(local)REAL for psger

DOUBLE REAL for pdger

Array, size at least (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)).

This array contains the entries of the distributed vector sub(*y*).

iy, *jy*
(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.

descy
(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy
(global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

a
(local)REAL for psger

DOUBLE REAL for pdger

Array, size (*lld_a*, LOCq(*ja*+*n*-1)).

	Before entry this array contains the local pieces of the distributed matrix sub(*A*).
ia, *ja*	(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

Output Parameters

a	Overwritten by the updated distributed matrix sub(*A*).

p?gerc

Performs a rank-1 update (conjugated) of a distributed general matrix.

Syntax

```
call pcgerc(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
call pzgerc(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

Include Files

- mkl_pblas.h

Description

The p?gerc routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*conjg(sub(y)') + sub(A),
```

where:

alpha is a scalar,

sub(*A*) is a *m*-by-*n* distributed general matrix, sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1),

sub(*x*) is an *m*-element distributed vector, sub(*y*) is an*n*-element distributed vector,

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*m*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*m*-1, *jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

m	(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*A*), *m*≥ 0.
n	(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*A*), *n*≥ 0.
alpha	(global)COMPLEX for pcgerc DOUBLE COMPLEX for pzgerc Specifies the scalar *alpha*.

x	(local)COMPLEX for pcgerc DOUBLE COMPLEX for pzgerc Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)). This array contains the entries of the distributed vector sub(*x*).
ix, *jx*	(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.
descx	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.
incx	(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.
y	(local)COMPLEX for pcgerc DOUBLE COMPLEX for pzgerc Array, size at least (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)). This array contains the entries of the distributed vector sub(*y*).
iy, *jy*	(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.
descy	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.
incy	(global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.
a	(local)COMPLEX for pcgerc DOUBLE COMPLEX for pzgerc Array, size at least (*lld_a*, LOCq(*ja*+*n*-1)). Before entry this array contains the local pieces of the distributed matrix sub(*A*).
ia, *ja*	(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

Output Parameters

a	Overwritten by the updated distributed matrix sub(*A*).

p?geru

Performs a rank-1 update (unconjugated) of a distributed general matrix.

Syntax

call pcgeru(*m*, *n*, *alpha*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*, *a*, *ia*, *ja*, *desca*)

call pzgeru(*m*, *n*, *alpha*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*, *a*, *ia*, *ja*, *desca*)

Include Files

- mkl_pblas.h

Description

The p?geru routines perform a matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*sub(y)' + sub(A),
```

where:

alpha is a scalar,

sub(*A*) is a *m*-by-*n* distributed general matrix, sub(*A*)=*A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1),

sub(*x*) is an *m*-element distributed vector, sub(*y*) is an *n*-element distributed vector,

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*m*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*m*-1, *jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

m

(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*A*), *m*≥ 0.

n

(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*A*), *n*≥ 0.

alpha

(global)COMPLEX for pcgeru

DOUBLE COMPLEX for pzgeru

Specifies the scalar *alpha*.

x

(local)COMPLEX for pcgeru

DOUBLE COMPLEX for pzgeru

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*

(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx

(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

y

(local)COMPLEX for pcgeru

DOUBLE COMPLEX for pzgeru

	Array, size at least `(jy-1)*m_y + iy+(n-1)*abs(incy))`. This array contains the entries of the distributed vector `sub(y)`.
iy, *jy*	(global) `INTEGER`. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix `sub(y)`, respectively.
descy	(global and local) `INTEGER` array of dimension 9. The array descriptor of the distributed matrix *Y*.
incy	(global) `INTEGER`. Specifies the increment for the elements of `sub(y)`. Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.
a	(local)`COMPLEX` for `pcgeru` `DOUBLE COMPLEX` for `pzgeru` Array, size at least `(lld_a, LOCq(ja+n-1))`. Before entry this array contains the local pieces of the distributed matrix `sub(A)`.
ia, *ja*	(global) `INTEGER`. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix `sub(A)`, respectively.
desca	(global and local) `INTEGER` array of dimension 9. The array descriptor of the distributed matrix *A*.

Output Parameters

a	Overwritten by the updated distributed matrix `sub(A)`.

p?hemv

Computes a distributed matrix-vector product using a Hermitian matrix.

Syntax

`call pchemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)`

`call pzhemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)`

Include Files

- `mkl_pblas.h`

Description

The `p?hemv` routines perform a distributed matrix-vector operation defined as

```
sub(y) := alpha*sub(A)*sub(x) + beta*sub(y),
```

where:

alpha and *beta* are scalars,

`sub(A)` is a *n*-by-*n* Hermitian distributed matrix, `sub(A)=A(ia:ia+n-1, ja:ja+n-1)` ,

`sub(x)` and `sub(y)` are distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix sub(*A*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*A*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*A*) is used.

n

(global) INTEGER. Specifies the order of the distributed matrix sub(*A*), $n \geq 0$.

alpha

(global)COMPLEX for pchemv

DOUBLE COMPLEX for pzhemv

Specifies the scalar *alpha*.

a

(local)COMPLEX for pchemv

DOUBLE COMPLEX for pzhemv

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

Before entry when *uplo* = 'U' or 'u', the *n*-by-*n* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and when *uplo* = 'L' or 'l', the *n*-by-*n* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x

(local)COMPLEX for pchemv

DOUBLE COMPLEX for pzhemv

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*

(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx	(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.
beta	(global)COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Specifies the scalar *beta*. When *beta* is set to zero, then sub(*y*) need not be set on input.
y	(local)COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Array, size at least (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)). This array contains the entries of the distributed vector sub(*y*).
iy, *jy*	(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.
descy	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.
incy	(global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

y	Overwritten by the updated distributed vector sub(*y*).

p?ahemv

Computes a distributed matrix-vector product using absolute values for a Hermitian matrix.

Syntax

```
call pcahemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

```
call pzahemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?ahemv routines perform a distributed matrix-vector operation defined as

```
sub(y) := abs(alpha)*abs(sub(A))*abs(sub(x)) + abs(beta*sub(y)),
```

where:

alpha and *beta* are scalars,

sub(*A*) is a *n*-by-*n* Hermitian distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) ,

sub(*x*) and sub(*y*) are distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix sub(*A*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*A*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*A*) is used.

n

(global) INTEGER. Specifies the order of the distributed matrix sub(*A*), *n*≥ 0.

alpha

(global)COMPLEX for pcahemv

DOUBLE COMPLEX for pzahemv

Specifies the scalar *alpha*.

a

(local)COMPLEX for pcahemv

DOUBLE COMPLEX for pzahemv

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

Before entry when *uplo* = 'U' or 'u', the *n*-by-*n* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and when *uplo* = 'L' or 'l', the *n*-by-*n* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x

(local)COMPLEX for pcahemv

DOUBLE COMPLEX for pzahemv

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*

(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx

(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

beta	(global)COMPLEX for pcahemv DOUBLE COMPLEX for pzahemv Specifies the scalar *beta*. When *beta* is set to zero, then sub(*y*) need not be set on input.
y	(local)COMPLEX for pcahemv DOUBLE COMPLEX for pzahemv Array, size at least (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)). This array contains the entries of the distributed vector sub(*y*).
iy, *jy*	(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.
descy	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.
incy	(global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

y	Overwritten by the updated distributed vector sub(*y*).

p?her

Performs a rank-1 update of a distributed Hermitian matrix.

Syntax

```
call pcher(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
call pzher(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
```

Include Files

- mkl_pblas.h

Description

The p?her routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*conjg(sub(x)') + sub(A),
```

where:

alpha is a real scalar,

sub(*A*) is a *n*-by-*n* distributed Hermitian matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1),

sub(*x*) is distributed vector.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1.

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix sub(*A*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*A*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*A*) is used.

n

(global) INTEGER. Specifies the order of the distributed matrix sub(*A*), $n \geq 0$.

alpha

(global)REAL for pcher

DOUBLE REAL for pzher

Specifies the scalar *alpha*.

x

(local)COMPLEX for pcher

DOUBLE COMPLEX for pzher

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*

(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx

(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

a

(local)COMPLEX for pcher

DOUBLE COMPLEX for pzher

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

Before entry with *uplo* = 'U' or 'u', the *n*-by-*n* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and with *uplo* = 'L' or 'l', the *n*-by-*n* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

Output Parameters

a — With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated distributed matrix sub(*A*).

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated distributed matrix sub(*A*).

p?her2

Performs a rank-2 update of a distributed Hermitian matrix.

Syntax

call pcher2(*uplo*, *n*, *alpha*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*, *a*, *ia*, *ja*, *desca*)

call pzher2(*uplo*, *n*, *alpha*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*, *a*, *ia*, *ja*, *desca*)

Include Files

- mkl_pblas.h

Description

The p?her2 routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*conj(sub(y)')+ conj(alpha)*sub(y)*conj(sub(x)') + sub(A),
```

where:

alpha is a scalar,

sub(*A*) is a *n*-by-*n* distributed Hermitian matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1),

sub(*x*) and sub(*y*) are distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

uplo — (global) CHARACTER*1. Specifies whether the upper or lower triangular part of the distributed Hermitian matrix sub(*A*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*A*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*A*) is used.

n — (global) INTEGER. Specifies the order of the distributed matrix sub(*A*), *n*≥ 0.

alpha — (global)COMPLEX for pcher2

DOUBLE COMPLEX for pzher2

Specifies the scalar *alpha*.

x (local)COMPLEX for pcher2

DOUBLE COMPLEX for pzher2

Array, size at least (*jx*-1)*m_*x* + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx* (global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx (global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

y (local)COMPLEX for pcher2

DOUBLE COMPLEX for pzher2

Array, size at least (*jy*-1)*m_*y* + *iy*+(*n*-1)*abs(*incy*)).

This array contains the entries of the distributed vector sub(*y*).

iy, *jy* (global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.

descy (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy (global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

a (local)COMPLEX for pcher2

DOUBLE COMPLEX for pzher2

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

Before entry with *uplo* = 'U' or 'u', the *n*-by-*n* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and with *uplo* = 'L' or 'l', the *n*-by-*n* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

Output Parameters

a — With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated distributed matrix sub(*A*).

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated distributed matrix sub(*A*).

p?symv

Computes a distributed matrix-vector product using a symmetric matrix.

Syntax

```
call pssymv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
call pdsymv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?symv routines perform a distributed matrix-vector operation defined as

```
sub(y)  := alpha*sub(A)*sub(x) + beta*sub(y),
```

where:

alpha and *beta* are scalars,

sub(*A*) is a *n*-by-*n* symmetric distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) ,

sub(*x*) and sub(*y*) are distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

uplo — (global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix sub(*A*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*A*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*A*) is used.

n — (global) INTEGER. Specifies the order of the distributed matrix sub(*A*), *n*≥ 0.

alpha — (global)REAL for pssymv

DOUBLE REAL for pdsymv

Specifies the scalar *alpha*.

a

(local)REAL for pssymv

DOUBLE REAL for pdsymv

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

Before entry when *uplo* = 'U' or 'u', the *n*-by-*n* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and when *uplo* = 'L' or 'l', the *n*-by-*n* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x

(local)REAL for pssymv

DOUBLE REAL for pdsymv

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*

(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx

(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

beta

(global)REAL for pssymv

DOUBLE REAL for pdsymv

Specifies the scalar *beta*. When *beta* is set to zero, then sub(*y*) need not be set on input.

y

(local)REAL for pssymv

DOUBLE REAL for pdsymv

Array, size at least (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)).

This array contains the entries of the distributed vector sub(*y*).

iy, *jy*

(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.

descy

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy (global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

y Overwritten by the updated distributed vector sub(*y*).

p?asymv

Computes a distributed matrix-vector product using absolute values for a symmetric matrix.

Syntax

call psasymv(*uplo*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

call pdasymv(*uplo*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

Include Files

- mkl_pblas.h

Description

The p?symv routines perform a distributed matrix-vector operation defined as

```
sub(y)  := abs(alpha)*abs(sub(A))*abs(sub(x)) + abs(beta*sub(y)),
```

where:

alpha and *beta* are scalars,

sub(*A*) is a *n*-by-*n* symmetric distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) ,

sub(*x*) and sub(*y*) are distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

uplo (global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix sub(*A*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*A*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*A*) is used.

n (global) INTEGER. Specifies the order of the distributed matrix sub(*A*), *n*≥ 0.

alpha (global)REAL for psasymv

DOUBLE REAL for pdasymv

Specifies the scalar *alpha*.

a (local)REAL for psasymv

DOUBLE REAL for pdasymv

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

Before entry when *uplo* = 'U' or 'u', the *n*-by-*n* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and when *uplo* = 'L' or 'l', the *n*-by-*n* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x

(local)REAL for psasymv

DOUBLE PRECISION for pdasymv

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*

(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx

(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

beta

(global)REAL for psasymv

DOUBLE PRECISION for pdasymv

Specifies the scalar *beta*. When *beta* is set to zero, then sub(*y*) need not be set on input.

y

(local)REAL for psasymv

DOUBLE PRECISION for pdasymv

Array, size at least (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)).

This array contains the entries of the distributed vector sub(*y*).

iy, *jy*

(global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.

descy

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy (global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

y Overwritten by the updated distributed vector sub(*y*).

p?syr

Performs a rank-1 update of a distributed symmetric matrix.

Syntax

```
call pssyr(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
call pdsyr(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
```

Include Files

- mkl_pblas.h

Description

The p?syr routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*sub(x)' + sub(A),
```

where:

alpha is a scalar,

sub(*A*) is a *n*-by-*n* distributed symmetric matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) ,

sub(*x*) is distributed vector.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

Input Parameters

uplo (global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix sub(*A*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*A*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*A*) is used.

n (global) INTEGER. Specifies the order of the distributed matrix sub(*A*), *n*≥ 0.

alpha (global)REAL for pssyr

DOUBLE REAL for pdsyr

Specifies the scalar *alpha*.

x (local)REAL for pssyr

DOUBLE REAL for pdsyr

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*

(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx

(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

a

(local)REAL for pssyr

DOUBLE REAL for pdsyr

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

Before entry with *uplo* = 'U' or 'u', the *n*-by-*n* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and with *uplo* = 'L' or 'l', the *n*-by-*n* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

Output Parameters

a

With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated distributed matrix sub(*A*).

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated distributed matrix sub(*A*).

p?syr2

Performs a rank-2 update of a distributed symmetric matrix.

Syntax

call pssyr2(*uplo*, *n*, *alpha*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*, *a*, *ia*, *ja*, *desca*)

call pdsyr2(*uplo*, *n*, *alpha*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*, *a*, *ia*, *ja*, *desca*)

Include Files

- mkl_pblas.h

Description

The p?syr2 routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*sub(y)'+ alpha*sub(y)*sub(x)' + sub(A),
```

where:

alpha is a scalar,

sub(*A*) is a *n*-by-*n* distributed symmetric matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) ,

sub(*x*) and sub(*y*) are distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the distributed symmetric matrix sub(*A*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*A*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*A*) is used.

n

(global) INTEGER. Specifies the order of the distributed matrix sub(*A*), *n*≥ 0.

alpha

(global)REAL for pssyr2

DOUBLE REAL for pdsyr2

Specifies the scalar *alpha*.

x

(local)REAL for pssyr2

DOUBLE REAL for pdsyr2

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*

(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx

(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

y

(local)REAL for pssyr2

DOUBLE REAL for pdsyr2

Array, size at least (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)).

This array contains the entries of the distributed vector sub(*y*).

iy, *jy* (global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.

descy (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy (global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

a (local)REAL for pssyr2

DOUBLE REAL for pdsyr2

Array, size (*lld_a*, LOCq(*ja*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

Before entry with *uplo* = 'U' or 'u', the *n*-by-*n* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the distributed symmetric matrix and the strictly lower triangular part of sub(*A*) is not referenced, and with *uplo* = 'L' or 'l', the *n*-by-*n* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the distributed symmetric matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

Output Parameters

a With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated distributed matrix sub(*A*).

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated distributed matrix sub(*A*).

p?trmv

Computes a distributed matrix-vector product using a triangular matrix.

Syntax

```
call pstrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pdtrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pctrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pztrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
```

Include Files

- mkl_pblas.h

Description

The p?trmv routines perform one of the following distributed matrix-vector operations defined as

sub(*x*) := sub(*A*)*sub(*x*), or sub(*x*) :=sub(*A*)'*sub(*x*), or sub(*x*) := conjg(sub(*A*)')*sub(*x*),

where:

sub(*A*) is a *n*-by-*n* unit, or non-unit, upper or lower triangular distributed matrix, sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1),

sub(*x*) is an *n*-element distributed vector.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the distributed matrix sub(*A*) is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans

(global) CHARACTER*1. Specifies the form of op(sub(*A*)) used in the matrix equation:

if *transa* = 'N' or 'n', then sub(*x*) := sub(*A*)*sub(*x*);

if *transa* = 'T' or 't', then sub(*x*) :=sub(*A*)'*sub(*x*);

if *transa* = 'C' or 'c', then sub(*x*) := conjg(sub(*A*)')*sub(*x*).

diag

(global) CHARACTER*1. Specifies whether the matrix sub(*A*) is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n

(global) INTEGER. Specifies the order of the distributed matrix sub(*A*), *n*≥0.

a

(local)REAL for pstrmv

DOUBLE PRECISION for pdtrmv

COMPLEX for pctrmv

DOUBLE COMPLEX for pztrmv

Array, size at least (*lld_a*, LOCq(1, *ja*+*n*-1)).

Before entry with *uplo* = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix sub(*A*) is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix sub(*A*) is not referenced .

When *diag* = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix sub(*A*) are not referenced either, but are assumed to be unity.

ia, *ja*
(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca
(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x
(local)REAL for pstrmv

DOUBLE PRECISION for pdtrmv

COMPLEX for pctrmv

DOUBLE COMPLEX for pztrmv

Array, size at least (*jx*-1)**m_x* + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*
(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx
(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx
(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

x
Overwritten by the transformed distributed vector sub(*x*).

p?atrmv

Computes a distributed matrix-vector product using absolute values for a triangular matrix.

Syntax

call psatrmv(*uplo*, *trans*, *diag*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

call pdatrmv(*uplo*, *trans*, *diag*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

call pcatrmv(*uplo*, *trans*, *diag*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

call pzatrmv(*uplo*, *trans*, *diag*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *x*, *ix*, *jx*, *descx*, *incx*, *beta*, *y*, *iy*, *jy*, *descy*, *incy*)

Include Files

- mkl_pblas.h

Description

The p?atrmv routines perform one of the following distributed matrix-vector operations defined as

sub(*y*) := abs(*alpha*)*abs(sub(*A*))*abs(sub(*x*))+ abs(*beta**sub(*y*)), or

sub(*y*) := abs(*alpha*)*abs(sub(*A*)')*abs(sub(*x*))+ abs(*beta**sub(*y*)), or

sub(*y*) := abs(*alpha*)*abs(conjg(sub(*A*)'))*abs(sub(*x*))+ abs(*beta**sub(*y*)),

where:

alpha and *beta* are scalars,

sub(*A*) is a *n*-by-*n* unit, or non-unit, upper or lower triangular distributed matrix, sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1),

sub(*x*) is an *n*-element distributed vector.

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1.

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the distributed matrix sub(*A*) is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans

(global) CHARACTER*1. Specifies the form of op(sub(*A*)) used in the matrix equation:

if *trans* = 'N' or 'n', then sub(*y*) := |*alpha*|*|sub(*A*)|*|sub(*x*)|+|*beta**sub(*y*)|;

if *trans* = 'T' or 't', then sub(*y*) := |*alpha*|*|sub(*A*)'|*|sub(*x*)|+|*beta**sub(*y*)|;

if *trans* = 'C' or 'c', then sub(*y*) := |*alpha*|*|conjg(sub(*A*)')|*|sub(*x*)|+|*beta**sub(*y*)|.

diag

(global) CHARACTER*1. Specifies whether the matrix sub(*A*) is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n

(global) INTEGER. Specifies the order of the distributed matrix sub(*A*), *n*≥0.

alpha

(global)REAL for psatrmv

DOUBLE PRECISION for pdatrmv

COMPLEX for pcatrmv

DOUBLE COMPLEX for pzatrmv

Specifies the scalar *alpha*.

a

(local)REAL for psatrmv

DOUBLE PRECISION for pdatrmv

COMPLEX for pcatrmv

DOUBLE COMPLEX for pzatrmv

Array, size at least (*lld_a*, LOCq(1, *ja*+*n*-1)).

Before entry with *uplo* = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix sub(*A*) is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix sub(*A*) is not referenced.

When *diag* = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix sub(*A*) are not referenced either, but are assumed to be unity.

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x

(local)REAL for psatrmv

DOUBLE PRECISION for pdatrmv

COMPLEX for pcatrmv

DOUBLE COMPLEX for pzatrmv

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector sub(*x*).

ix, *jx*

(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

incx

(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

beta

(global)REAL for psatrmv

DOUBLE PRECISION for pdatrmv

COMPLEX for pcatrmv

DOUBLE COMPLEX for pzatrmv

Specifies the scalar *beta*. When *beta* is set to zero, then sub(*y*) need not be set on input.

y

(local)REAL for psatrmv

DOUBLE PRECISION for pdatrmv

COMPLEX for pcatrmv

DOUBLE COMPLEX for pzatrmv

Array, size (*jy*-1)*m_y + *iy*+(*m*-1)*abs(*incy*)) when *trans* = 'N' or 'n', and (*jy*-1)*m_y + *iy*+(*n*-1)*abs(*incy*)) otherwise.

This array contains the entries of the distributed vector sub(*y*).

iy, *jy* — (global) INTEGER. The row and column indices in the distributed matrix *Y* indicating the first row and the first column of the submatrix sub(*y*), respectively.

descy — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy — (global) INTEGER. Specifies the increment for the elements of sub(*y*). Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

y — Overwritten by the transformed distributed vector sub(*y*).

p?trsv

Solves a system of linear equations whose coefficients are in a distributed triangular matrix.

Syntax

```
call pstrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pdtrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pctrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pztrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
```

Include Files

- mkl_pblas.h

Description

The p?trsv routines solve one of the systems of equations:

sub(*A*)*sub(*x*) = *b*, or sub(*A*)'*sub(*x*) = *b*, or conjg(sub(*A*)')*sub(*x*) = *b*,

where:

sub(*A*) is a *n*-by-*n* unit, or non-unit, upper or lower triangular distributed matrix, sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1),

b and sub(*x*) are *n*-element distributed vectors,

sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,.

The routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

Input Parameters

uplo — (global) CHARACTER*1. Specifies whether the distributed matrix sub(*A*) is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans (global) CHARACTER*1. Specifies the form of the system of equations:

if *transa* = 'N' or 'n', then sub(*A*)*sub(*x*) = *b*;

if *transa* = 'T' or 't', then sub(*A*)'*sub(*x*) = *b*;

if *transa* = 'C' or 'c', then conjg(sub(*A*)')*sub(*x*) = *b*.

diag (global) CHARACTER*1. Specifies whether the matrix sub(*A*) is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n (global) INTEGER. Specifies the order of the distributed matrix sub(*A*), *n*≥ 0.

a (local)REAL for pstrsv

DOUBLE PRECISION for pdtrsv

COMPLEX for pctrsv

DOUBLE COMPLEX for pztrsv

Array, size at least (*lld_a*, LOCq(1, *ja*+*n*-1)).

Before entry with *uplo* = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix sub(*A*) is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix sub(*A*) is not referenced .

When *diag* = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix sub(*A*) are not referenced either, but are assumed to be unity.

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x (local)REAL for pstrsv

DOUBLE PRECISION for pdtrsv

COMPLEX for pctrsv

DOUBLE COMPLEX for pztrsv

Array, size at least (*jx*-1)*m_x + *ix*+(*n*-1)*abs(*incx*)).

	This array contains the entries of the distributed vector sub(*x*). Before entry, sub(*x*) must contain the *n*-element right-hand side distributed vector *b*.
ix, *jx*	(global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix sub(*x*), respectively.
descx	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.
incx	(global) INTEGER. Specifies the increment for the elements of sub(*x*). Only two values are supported, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

x	Overwritten with the solution vector.

PBLAS Level 3 Routines

The PBLAS Level 3 routines perform distributed matrix-matrix operations. Table "PBLAS Level 3 Routine Groups and Their Data Types" lists the PBLAS Level 3 routine groups and the data types associated with them.

PBLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
p?geadd	s, d, c, z	Distributed matrix-matrix sum of general matrices
p?tradd	s, d, c, z	Distributed matrix-matrix sum of triangular matrices
p?gemm	s, d, c, z	Distributed matrix-matrix product of general matrices
p?hemm	c, z	Distributed matrix-matrix product, one matrix is Hermitian
p?herk	c, z	Rank-k update of a distributed Hermitian matrix
p?her2k	c, z	Rank-2k update of a distributed Hermitian matrix
p?symm	s, d, c, z	Matrix-matrix product of distributed symmetric matrices
p?syrk	s, d, c, z	Rank-k update of a distributed symmetric matrix
p?syr2k	s, d, c, z	Rank-2k update of a distributed symmetric matrix
p?tran	s, d	Transposition of a real distributed matrix
p?tranc	c, z	Transposition of a complex distributed matrix (conjugated)
p?tranu	c, z	Transposition of a complex distributed matrix
p?trmm	s, d, c, z	Distributed matrix-matrix product, one matrix is triangular
p?trsm	s, d, c, z	Solution of a distributed matrix equation, one matrix is triangular

p?geadd

Performs sum operation for two distributed general matrices.

Syntax

```
call psgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pdgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pcgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pzgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?geadd routines perform sum operation for two distributed general matrices. The operation is defined as

```
sub(C):=beta*sub(C) + alpha*op(sub(A)),
```

where:

op(*x*) is one of op(*x*) = *x*, or op(*x*) = *x*',

alpha and *beta* are scalars,

sub(*C*) is an *m*-by-*n* distributed matrix, sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*m*-1).

Input Parameters

trans

(global) CHARACTER*1. Specifies the operation:

if *trans* = 'N' or 'n', then op(sub(*A*)) := sub(*A*);

if *trans* = 'T' or 't', then op(sub(*A*)) := sub(*A*)';

if *trans* = 'C' or 'c', then op(sub(*A*)) := sub(*A*)'.

m

(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*C*) and the number of columns of the submatrix sub(*A*), $m \geq 0$.

n

(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*C*) and the number of rows of the submatrix sub(*A*), $n \geq 0$.

alpha

(global)REAL for psgeadd

DOUBLE PRECISION for pdgeadd

COMPLEX for pcgeadd

DOUBLE COMPLEX for pzgeadd

Specifies the scalar *alpha*.

a

(local)REAL for psgeadd

DOUBLE PRECISION for pdgeadd

COMPLEX for pcgeadd

DOUBLE COMPLEX for pzgeadd

Array, size (*lld_a*, LOCq(*ja*+*m*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

ia, *ja* — (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

beta — (global)REAL for psgeadd

DOUBLE PRECISION for pdgeadd

COMPLEX for pcgeadd

DOUBLE COMPLEX for pzgeadd

Specifies the scalar *beta*.

When *beta* is equal to zero, then sub(*C*) need not be set on input.

c — (local)REAL for psgeadd

DOUBLE PRECISION for pdgeadd

COMPLEX for pcgeadd

DOUBLE COMPLEX for pzgeadd

Array, size (*lld_c*, LOCq(*jc*+*n*-1)).

This array contains the local pieces of the distributed matrix sub(*C*).

ic, *jc* — (global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c — Overwritten by the updated submatrix.

p?tradd

Performs sum operation for two distributed triangular matrices.

Syntax

```
call pstradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pdtradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pctradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pztradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?tradd routines perform sum operation for two distributed triangular matrices. The operation is defined as

```
sub(C):=beta*sub(C) + alpha*op(sub(A)),
```

where:

op(*x*) is one of op(*x*) = *x*, or op(*x*) = *x*', or op(*x*) = conjg(*x*').

alpha and *beta* are scalars,

sub(*C*) is an *m*-by-*n* distributed matrix, sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*m*-1).

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the distributed matrix sub(*C*) is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans

(global) CHARACTER*1. Specifies the operation:

if *trans* = 'N' or 'n', then op(sub(*A*)) := sub(*A*);

if *trans* = 'T' or 't', then op(sub(*A*)) := sub(*A*)';

if *trans* = 'C' or 'c', then op(sub(*A*)) := conjg(sub(*A*)').

m

(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*C*) and the number of columns of the submatrix sub(*A*), $m \geq 0$.

n

(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*C*) and the number of rows of the submatrix sub(*A*), $n \geq 0$.

alpha

(global)REAL for pstradd

DOUBLE PRECISION for pdtradd

COMPLEX for pctradd

DOUBLE COMPLEX for pztradd

Specifies the scalar *alpha*.

a

(local)REAL for pstradd

DOUBLE PRECISION for pdtradd

COMPLEX for pctradd

DOUBLE COMPLEX for pztradd

Array, size (*lld_a*, LOCq(*ja*+*m*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

ia, *ja*	(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.
beta	(global)REAL for pstradd DOUBLE PRECISION for pdtradd COMPLEX for pctradd DOUBLE COMPLEX for pztradd Specifies the scalar *beta*. When *beta* is equal to zero, then sub(*C*) need not be set on input.
c	(local)REAL for pstradd DOUBLE PRECISION for pdtradd COMPLEX for pctradd DOUBLE COMPLEX for pztradd Array, size (*lld_c*, LOCq(*jc*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*C*).
ic, *jc*	(global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.
descc	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c	Overwritten by the updated submatrix.

p?gemm

Computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product for distributed matrices.

Syntax

call psgemm(*transa*, *transb*, *m*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pdgemm(*transa*, *transb*, *m*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pcgemm(*transa*, *transb*, *m*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pzgemm(*transa*, *transb*, *m*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

Include Files

- mkl_pblas.h

Description

The p?gemm routines perform a matrix-matrix operation with general distributed matrices. The operation is defined as

```
sub(C) := alpha*op(sub(A))*op(sub(B)) + beta*sub(C),
```

where:

op(*x*) is one of op(*x*) = *x*, or op(*x*) = *x*',

alpha and *beta* are scalars,

sub(*A*)=*A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*k*-1), sub(*B*)=*B*(*ib*:*ib*+*k*-1, *jb*:*jb*+*n*-1), and sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1), are distributed matrices.

Input Parameters

transa

(global) CHARACTER*1. Specifies the form of op(sub(*A*)) used in the matrix multiplication:

if *transa* = 'N' or 'n', then op(sub(*A*)) = sub(*A*);

if *transa* = 'T' or 't', then op(sub(*A*)) = sub(*A*)';

if *transa* = 'C' or 'c', then op(sub(*A*)) = sub(*A*)'.

transb

(global) CHARACTER*1. Specifies the form of op(sub(*B*)) used in the matrix multiplication:

if *transb* = 'N' or 'n', then op(sub(*B*)) = sub(*B*);

if *transb* = 'T' or 't', then op(sub(*B*)) = sub(*B*)';

if *transb* = 'C' or 'c', then op(sub(*B*)) = sub(*B*)'.

m

(global) INTEGER. Specifies the number of rows of the distributed matrices op(sub(*A*)) and sub(*C*), $m \geq 0$.

n

(global) INTEGER. Specifies the number of columns of the distributed matrices op(sub(*B*)) and sub(*C*), $n \geq 0$.

The value of *n* must be at least zero.

k

(global) INTEGER. Specifies the number of columns of the distributed matrix op(sub(*A*)) and the number of rows of the distributed matrix op(sub(*B*)).

The value of *k* must be greater than or equal to 0.

alpha

(global)REAL for psgemm

DOUBLE PRECISION for pdgemm

COMPLEX for pcgemm

DOUBLE COMPLEX for pzgemm

Specifies the scalar *alpha*.

When *alpha* is equal to zero, then the local entries of the arrays *a* and *b* corresponding to the entries of the submatrices sub(*A*) and sub(*B*) respectively need not be set on input.

a (local) REAL for psgemm

DOUBLE PRECISION for pdgemm

COMPLEX for pcgemm

DOUBLE COMPLEX for pzgemm

Array, size *lld_a* by *kla*, where *kla* is LOCc(*ja*+*k*-1) when *transa* = 'N' or 'n', and is LOCq(*ja*+*m*-1) otherwise. Before entry this array must contain the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

b (local)REAL for psgemm

DOUBLE PRECISION for pdgemm

COMPLEX for pcgemm

DOUBLE COMPLEX for pzgemm

Array, size *lld_b* by *klb*, where *klb* is LOCc(*jb*+*n*-1) when *transb* = 'N' or 'n', and is LOCq(*jb*+*k*-1) otherwise. Before entry this array must contain the local pieces of the distributed matrix sub(*B*).

ib, *jb* (global) INTEGER. The row and column indices in the distributed matrix *B* indicating the first row and the first column of the submatrix sub(*B*), respectively

descb (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *B*.

beta (global)REAL for psgemm

DOUBLE PRECISION for pdgemm

COMPLEX for pcgemm

DOUBLE COMPLEX for pzgemm

Specifies the scalar *beta*.

When *beta* is equal to zero, then sub(*C*) need not be set on input.

c (local)REAL for psgemm

DOUBLE PRECISION for pdgemm

COMPLEX for pcgemm

DOUBLE COMPLEX for pzgemm

Array, size (*lld_a*, LOCq(*jc*+*n*-1)). Before entry this array must contain the local pieces of the distributed matrix sub(*C*).

ic, *jc* (global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively

descc (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c Overwritten by the *m*-by-*n* distributed matrix *alpha**op(sub(*A*))*op(sub(*B*)) + *beta**sub(*C*).

p?hemm

Performs a scalar-matrix-matrix product (one matrix operand is Hermitian) and adds the result to a scalar-matrix product.

Syntax

call pchemm(*side*, *uplo*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pzhemm(*side*, *uplo*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

Include Files

- mkl_pblas.h

Description

The p?hemm routines perform a matrix-matrix operation with distributed matrices. The operation is defined as

sub(*C*):=*alpha**sub(*A*)*sub(*B*)+ *beta**sub(*C*),

or

sub(*C*):=*alpha**sub(*B*)*sub(*A*)+ *beta**sub(*C*),

where:

alpha and *beta* are scalars,

sub(*A*) is a Hermitian distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*m*-1), if *side* = 'L', and sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), if *side* = 'R'.

sub(*B*) and sub(*C*) are *m*-by-*n* distributed matrices.

sub(*B*)=*B*(*ib*:*ib*+*m*-1, *jb*:*jb*+*n*-1), sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1).

Input Parameters

side (global) CHARACTER*1. Specifies whether the Hermitian distributed matrix sub(*A*) appears on the left or right in the operation:

if *side* = 'L' or 'l', then sub(*C*) := *alpha**sub(*A*) *sub(*B*) + *beta**sub(*C*);

if *side* = 'R' or 'r', then sub(*C*) := *alpha**sub(*B*) *sub(*A*) + *beta**sub(*C*).

uplo (global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix sub(*A*) is used:

if *uplo* = 'U' or 'u', then the upper triangular part is used;

if *uplo* = 'L' or 'l', then the lower triangular part is used.

m (global) INTEGER. Specifies the number of rows of the distribute submatrix sub(*C*), $m \geq 0$.

n (global) INTEGER. Specifies the number of columns of the distribute submatrix sub(*C*), $n \geq 0$.

alpha (global)COMPLEX for pchemm

DOUBLE COMPLEX for pzhemm

Specifies the scalar *alpha*.

a (local)COMPLEX for pchemm

DOUBLE COMPLEX for pzhemm

Array, size (*lld_a*, LOCq(*ja*+*na*-1)).

Before entry this array must contain the local pieces of the symmetric distributed matrix sub(*A*), such that when *uplo* = 'U' or 'u', the *na*-by-*na* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and when *uplo* = 'L' or 'l', the *na*-by-*na* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

b (local)COMPLEX for pchemm

DOUBLE COMPLEX for pzhemm

Array, size (*lld_b*, LOCq(*jb*+*n*-1)). Before entry this array must contain the local pieces of the distributed matrix sub(*B*).

ib, *jb* (global) INTEGER. The row and column indices in the distributed matrix *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *B*.

beta (global)COMPLEX for pchemm

DOUBLE COMPLEX for pzhemm

Specifies the scalar *beta*.

When *beta* is set to zero, then sub(*C*) need not be set on input.

c	(local)COMPLEX for pchemm DOUBLE COMPLEX for pzhemm Array, size (*lld_c*, LOCq(*jc*+*n*-1)). Before entry this array must contain the local pieces of the distributed matrix sub(*C*).
ic, *jc*	(global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively
descc	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c	Overwritten by the *m*-by-*n* updated distributed matrix.

p?herk

Performs a rank-k update of a distributed Hermitian matrix.

Syntax

```
call pcherk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pzherk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?herk routines perform a distributed matrix-matrix operation defined as

```
sub(C):=alpha*sub(A)*conjg(sub(A)')+ beta*sub(C),
```

or

```
sub(C):=alpha*conjg(sub(A)')*sub(A)+ beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(*C*) is an *n*-by-*n* Hermitian distributed matrix, sub(*C*)=C(*ic*:*ic*+*n*-1, *jc*:*jc*+*n*-1).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*k*-1), if *trans* = 'N' or 'n', and sub(*A*)=*A*(*ia*:*ia*+*k*-1, *ja*:*ja*+*n*-1) otherwise.

Input Parameters

uplo	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix sub(*C*) is used: If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*C*) is used. If *uplo* = 'L' or 'l', then the low triangular part of the sub(*C*) is used.
trans	(global) CHARACTER*1. Specifies the operation:

if *trans* = 'N' or 'n', then sub(*C*) := *alpha**sub(*A*)*conjg(sub(*A*)') + *beta**sub(*C*);

if *trans* = 'C' or 'c', then sub(*C*) := *alpha**conjg(sub(*A*)')*sub(*A*) + *beta**sub(*C*).

n (global) INTEGER. Specifies the order of the distributed matrix sub(*C*), *n*≥ 0.

k (global) INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the distributed matrix sub(*A*) , and on entry with *trans* = 'T' or 't' or 'C' or 'c', *k* specifies the number of rows of the distributed matrix sub(*A*), *k*≥ 0.

alpha (global)REAL for pcherk

DOUBLE PRECISION for pzherk

Specifies the scalar *alpha*.

a (local)COMPLEX for pcherk

DOUBLE COMPLEX for pzherk

Array, size (*lld_a*, *kla*), where *kla* is LOCq(*ja*+*k*-1) when *trans* = 'N' or 'n', and is LOCq(*ja*+*n*-1) otherwise. Before entry with *trans* = 'N' or 'n', this array contains the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

beta (global)REAL for pcherk

DOUBLE PRECISION for pzherk

Specifies the scalar *beta*.

c (local)COMPLEX for pcherk

DOUBLE COMPLEX for pzherk

Array, size (*lld_c*, LOCq(*jc*+*n*-1)).

Before entry with *uplo* = 'U' or 'u', this array contains *n*-by-*n* upper triangular part of the symmetric distributed matrix sub(*C*) and its strictly lower triangular part is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains *n*-by-*n* lower triangular part of the symmetric distributed matrix sub(*C*) and its strictly upper triangular part is not referenced.

ic, *jc* (global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c — With *uplo* = 'U' or 'u', the upper triangular part of sub(*C*) is overwritten by the upper triangular part of the updated distributed matrix.

With *uplo* = 'L' or 'l', the lower triangular part of sub(*C*) is overwritten by the upper triangular part of the updated distributed matrix.

p?her2k

Performs a rank-2k update of a Hermitian distributed matrix.

Syntax

call pcher2k(*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pzher2k(*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

Include Files

- mkl_pblas.h

Description

The p?her2k routines perform a distributed matrix-matrix operation defined as

sub(*C*):=*alpha**sub(*A*)*conjg(sub(*B*)')+ conjg(*alpha*)*sub(*B*)*conjg(sub(*A*)')+*beta**sub(*C*),

or

sub(*C*):=*alpha**conjg(sub(*A*)')*sub(*A*)+ conjg(*alpha*)*conjg(sub(*B*)')*sub(*A*) + *beta**sub(*C*),

where:

alpha and *beta* are scalars,

sub(*C*) is an *n*-by-*n* Hermitian distributed matrix, sub(*C*) = C(*ic*:*ic*+*n*-1, *jc*:*jc*+*n*-1).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*k*-1), if *trans* = 'N' or 'n', and sub(*A*) = *A*(*ia*:*ia*+*k*-1, *ja*:*ja*+*n*-1) otherwise.

sub(*B*) is a distributed matrix, sub(*B*) = *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*k*-1), if *trans* = 'N' or 'n', and sub(*B*)=*B*(*ib*:*ib*+*k*-1, *jb*:*jb*+*n*-1) otherwise.

Input Parameters

uplo — (global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix sub(*C*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*C*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*C*) is used.

trans — (global) CHARACTER*1. Specifies the operation:

if *trans* = 'N' or 'n', then sub(*C*) := *alpha**sub(*A*)*conjg(sub(*B*)') + conjg(*alpha*)*sub(*B*)*conjg(sub(*A*)') + *beta**sub(*C*);

if *trans* = 'C' or 'c', then sub(*C*) := *alpha**conjg(sub(*A*)')*sub(*A*) + conjg(*alpha*)*conjg(sub(*B*)')*sub(*A*) + *beta**sub(*C*).

n

(global) INTEGER. Specifies the order of the distributed matrix sub(*C*), $n \geq 0$.

k

(global) INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the distributed matrices sub(*A*) and sub(*B*), and on entry with *trans* = 'C' or 'c' , *k* specifies the number of rows of the distributed matrices sub(*A*) and sub(*B*), $k \geq 0$.

alpha

(global)COMPLEX for pcher2k

DOUBLE COMPLEX for pzher2k

Specifies the scalar *alpha*.

a

(local)COMPLEX for pcher2k

DOUBLE COMPLEX for pzher2k

Array, size (*lld_a*, *kla*), where *kla* is LOCq(*ja*+*k*-1) when *trans* = 'N' or 'n', and is LOCq(*ja*+*n*-1) otherwise. Before entry with *trans* = 'N' or 'n', this array contains the local pieces of the distributed matrix sub(*A*).

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

b

(local)COMPLEX for pcher2k

DOUBLE COMPLEX for pzher2k

Array, size (*lld_b*, *klb*), where *klb* is LOCq(*jb*+*k*-1) when *trans* = 'N' or 'n', and is LOCq(*jb*+*n*-1) otherwise. Before entry with *trans* = 'N' or 'n', this array contains the local pieces of the distributed matrix sub(*B*).

ib, *jb*

(global) INTEGER. The row and column indices in the distributed matrix *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *B*.

beta

(global)REAL for pcher2k

DOUBLE PRECISION for pzher2k

Specifies the scalar *beta*.

c

(local)COMPLEX for pcher2k

DOUBLE COMPLEX for pzher2k

Array, size (*lld_c*, LOCq(*jc*+*n*-1)).

Before entry with *uplo* = 'U' or 'u', this array contains *n*-by-*n* upper triangular part of the symmetric distributed matrix sub(*C*) and its strictly lower triangular part is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains *n*-by-*n* lower triangular part of the symmetric distributed matrix sub(*C*) and its strictly upper triangular part is not referenced.

ic, *jc*

(global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c

With *uplo* = 'U' or 'u', the upper triangular part of sub(*C*) is overwritten by the upper triangular part of the updated distributed matrix.

With *uplo* = 'L' or 'l', the lower triangular part of sub(*C*) is overwritten by the upper triangular part of the updated distributed matrix.

p?symm

Performs a scalar-matrix-matrix product (one matrix operand is symmetric) and adds the result to a scalar-matrix product for distribute matrices.

Syntax

call pssymm(*side*, *uplo*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pdsymm(*side*, *uplo*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pcsymm(*side*, *uplo*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pzsymm(*side*, *uplo*, *m*, *n*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

Include Files

- mkl_pblas.h

Description

The p?symm routines perform a matrix-matrix operation with distributed matrices. The operation is defined as

sub(*C*):=*alpha**sub(*A*)*sub(*B*)+ *beta**sub(*C*),

or

sub(*C*):=*alpha**sub(*B*)*sub(*A*)+ *beta**sub(*C*),

where:

alpha and *beta* are scalars,

sub(*A*) is a symmetric distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*m*-1), if *side* ='L', and sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), if *side* ='R'.

sub(*B*) and sub(*C*) are *m*-by-*n* distributed matrices.

sub(*B*)=*B*(*ib*:*ib*+*m*-1, *jb*:*jb*+*n*-1), sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1).

Input Parameters

side

(global) CHARACTER*1. Specifies whether the symmetric distributed matrix sub(*A*) appears on the left or right in the operation:

if *side* = 'L' or 'l', then sub(*C*) := *alpha**sub(*A*) *sub(*B*) + *beta**sub(*C*);

if *side* = 'R' or 'r', then sub(*C*) := *alpha**sub(*B*) *sub(*A*) + *beta**sub(*C*).

uplo

(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix sub(*A*) is used:

if *uplo* = 'U' or 'u', then the upper triangular part is used;

if *uplo* = 'L' or 'l', then the lower triangular part is used.

m

(global) INTEGER. Specifies the number of rows of the distribute submatrix sub(*C*), $m \geq 0$.

n

(global) INTEGER. Specifies the number of columns of the distribute submatrix sub(*C*), $m \geq 0$.

alpha

(global)REAL for pssymm

DOUBLE PRECISION for pdsymm

COMPLEX for pcsymm

DOUBLE COMPLEX for pzsymm

Specifies the scalar *alpha*.

a

(local)REAL for pssymm

DOUBLE PRECISION for pdsymm

COMPLEX for pcsymm

DOUBLE COMPLEX for pzsymm

Array, size (*lld_a*, LOCq(*ja*+*na*-1)).

Before entry this array must contain the local pieces of the symmetric distributed matrix sub(*A*), such that when *uplo* = 'U' or 'u', the *na*-by-*na* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and when *uplo* = 'L' or 'l', the *na*-by-*na* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

b

(local)REAL for pssymm

DOUBLE PRECISION for pdsymm

COMPLEX for pcsymm

DOUBLE COMPLEX for pzsymm

Array, size (*lld_b*, LOCq(*jb*+*n*-1)). Before entry this array must contain the local pieces of the distributed matrix sub(*B*).

ib, *jb* — (global) INTEGER. The row and column indices in the distributed matrix *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *B*.

beta — (global)REAL for pssymm

DOUBLE PRECISION for pdsymm

COMPLEX for pcsymm

DOUBLE COMPLEX for pzsymm

Specifies the scalar *beta*.

When *beta* is set to zero, then sub(*C*) need not be set on input.

c — (local)REAL for pssymm

DOUBLE PRECISION for pdsymm

COMPLEX for pcsymm

DOUBLE COMPLEX for pzsymm

Array, size (*lld_c*, LOCq(*jc*+*n*-1)). Before entry this array must contain the local pieces of the distributed matrix sub(*C*).

ic, *jc* — (global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c — Overwritten by the *m*-by-*n* updated matrix.

p?syrk

Performs a rank-k update of a symmetric distributed matrix.

Syntax

```
call pssyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pdsyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pcsyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pzsyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?syrk routines perform a distributed matrix-matrix operation defined as

```
sub(C):=alpha*sub(A)*sub(A)'+ beta*sub(C),
```

or

```
sub(C):=alpha*sub(A)'*sub(A)+ beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(*C*) is an *n*-by-*n* symmetric distributed matrix, sub(*C*)=C(*ic*:*ic*+*n*-1, *jc*:*jc*+*n*-1).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*k*-1), if *trans* = 'N' or 'n', and sub(*A*)=*A*(*ia*:*ia*+*k*-1, *ja*:*ja*+*n*-1) otherwise.

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix sub(*C*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*C*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*C*) is used.

trans

(global) CHARACTER*1. Specifies the operation:

if *trans* = 'N' or 'n', then sub(*C*) := *alpha**sub(*A*)*sub(*A*)' + *beta**sub(*C*);

if *trans* = 'T' or 't', then sub(*C*) := *alpha**sub(*A*)'*sub(*A*) + *beta**sub(*C*).

n

(global) INTEGER. Specifies the order of the distributed matrix sub(*C*), $n \geq 0$.

k

(global) INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the distributed matrix sub(*A*) , and on entry with *trans* = 'T' or 't' , *k* specifies the number of rows of the distributed matrix sub(*A*), $k \geq 0$.

alpha

(global)REAL for pssyrk

DOUBLE PRECISION for pdsyrk

COMPLEX for pcsyrk

DOUBLE COMPLEX for pzsyrk

Specifies the scalar *alpha*.

a

(local)REAL for pssyrk

DOUBLE PRECISION for pdsyrk

COMPLEX for pcsyrk

DOUBLE COMPLEX for pzsyrk

Array, size (*lld_a*, *kla*), where *kla* is LOCq(*ja*+*k*-1) when *trans* = 'N' or 'n', and is LOCq(*ja*+*n*-1) otherwise. Before entry with *trans* = 'N' or 'n', this array contains the local pieces of the distributed matrix sub(*A*).

ia, *ja* — (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

beta — (global)REAL for pssyrk

DOUBLE PRECISION for pdsyrk

COMPLEX for pcsyrk

DOUBLE COMPLEX for pzsyrk

Specifies the scalar *beta*.

c — (local)REAL for pssyrk

DOUBLE PRECISION for pdsyrk

COMPLEX for pcsyrk

DOUBLE COMPLEX for pzsyrk

Array, size (*lld_c*, LOCq(*jc*+*n*-1)).

Before entry with *uplo* = 'U' or 'u', this array contains *n*-by-*n* upper triangular part of the symmetric distributed matrix sub(*C*) and its strictly lower triangular part is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains *n*-by-*n* lower triangular part of the symmetric distributed matrix sub(*C*) and its strictly upper triangular part is not referenced.

ic, *jc* — (global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c — With *uplo* = 'U' or 'u', the upper triangular part of sub(*C*) is overwritten by the upper triangular part of the updated distributed matrix.

With *uplo* = 'L' or 'l', the lower triangular part of sub(*C*) is overwritten by the upper triangular part of the updated distributed matrix.

p?syr2k

Performs a rank-2k update of a symmetric distributed matrix.

Syntax

call pssyr2k(*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pdsyr2k(*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pcsyr2k(*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

call pzsyr2k(*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *beta*, *c*, *ic*, *jc*, *descc*)

Include Files

- mkl_pblas.h

Description

The p?syr2k routines perform a distributed matrix-matrix operation defined as

sub(*C*):=*alpha**sub(*A*)*sub(*B*)'+*alpha**sub(*B*)*sub(*A*)'+ *beta**sub(*C*),

or

sub(*C*):=*alpha**sub(*A*)'*sub(*B*) +*alpha**sub(*B*)'*sub(*A*) + *beta**sub(*C*),

where:

alpha and *beta* are scalars,

sub(*C*) is an *n*-by-*n* symmetric distributed matrix, sub(*C*)=C(*ic*:*ic*+*n*-1, *jc*:*jc*+*n*-1).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*k*-1), if *trans* = 'N' or 'n', and sub(*A*)=*A*(*ia*:*ia*+*k*-1, *ja*:*ja*+*n*-1) otherwise.

sub(*B*) is a distributed matrix, sub(*B*)=*B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*k*-1), if *trans* = 'N' or 'n', and sub(*B*)=*B*(*ib*:*ib*+*k*-1, *jb*:*jb*+*n*-1) otherwise.

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix sub(*C*) is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the sub(*C*) is used.

If *uplo* = 'L' or 'l', then the low triangular part of the sub(*C*) is used.

trans

(global) CHARACTER*1. Specifies the operation:

if *trans* = 'N' or 'n', then sub(*C*) := *alpha**sub(*A*)*sub(*B*)' + *alpha**sub(*B*)*sub(*A*)' + *beta**sub(*C*);

if *trans* = 'T' or 't', then sub(*C*) := *alpha**sub(*B*)'*sub(*A*) + *alpha**sub(*A*)'*sub(*B*) + *beta**sub(*C*).

n

(global) INTEGER. Specifies the order of the distributed matrix sub(*C*), *n*≥ 0.

k (global) INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the distributed matrices sub(*A*) and sub(*B*), and on entry with *trans* = 'T' or 't' , *k* specifies the number of rows of the distributed matrices sub(*A*) and sub(*B*), *k*≥ 0.

alpha (global)REAL for pssyr2k

DOUBLE PRECISION for pdsyr2k

COMPLEX for pcsyr2k

DOUBLE COMPLEX for pzsyr2k

Specifies the scalar *alpha*.

a (local)REAL for pssyr2k

DOUBLE PRECISION for pdsyr2k

COMPLEX for pcsyr2k

DOUBLE COMPLEX for pzsyr2k

Array, size (*lld_a*, *kla*), where *kla* is LOCq(*ja*+*k*–1) when *trans* = 'N' or 'n', and is LOCq(*ja*+*n*–1) otherwise. Before entry with *trans* = 'N' or 'n', this array contains the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

b (local)REAL for pssyr2k

DOUBLE PRECISION for pdsyr2k

COMPLEX for pcsyr2k

DOUBLE COMPLEX for pzsyr2k

Array, size (*lld_b*, *klb*), where *klb* is LOCq(*jb*+*k*–1) when *trans* = 'N' or 'n', and is LOCq(*jb*+*n*–1) otherwise. Before entry with *trans* = 'N' or 'n', this array contains the local pieces of the distributed matrix sub(*B*).

ib, *jb* (global) INTEGER. The row and column indices in the distributed matrix *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *B*.

beta (global)REAL for pssyr2k

DOUBLE PRECISION for pdsyr2k

COMPLEX for pcsyr2k

DOUBLE COMPLEX for pzsyr2k

Specifies the scalar *beta*.

c (local)REAL for pssyr2k

	DOUBLE PRECISION for pdsyr2k COMPLEX for pcsyr2k DOUBLE COMPLEX for pzsyr2k Array, size (*lld_c*, LOCq(*jc*+*n*-1)). Before entry with *uplo* = 'U' or 'u', this array contains *n*-by-*n* upper triangular part of the symmetric distributed matrix sub(*C*) and its strictly lower triangular part is not referenced. Before entry with *uplo* = 'L' or 'l', this array contains *n*-by-*n* lower triangular part of the symmetric distributed matrix sub(*C*) and its strictly upper triangular part is not referenced.
ic, *jc*	(global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.
descc	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c	With *uplo* = 'U' or 'u', the upper triangular part of sub(*C*) is overwritten by the upper triangular part of the updated distributed matrix. With *uplo* = 'L' or 'l', the lower triangular part of sub(*C*) is overwritten by the upper triangular part of the updated distributed matrix.

p?tran

Transposes a real distributed matrix.

Syntax

```
call pstran(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pdtran(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?tran routines transpose a real distributed matrix. The operation is defined as

```
sub(C):=beta*sub(C) + alpha*sub(A)',
```

where:

alpha and *beta* are scalars,

sub(*C*) is an *m*-by-*n* distributed matrix, sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*m*-1).

Input Parameters

m	(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*C*), *m*≥ 0.
n	(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*C*) , *n*≥ 0.
alpha	(global)REAL for pstran DOUBLE PRECISION for pdtran Specifies the scalar *alpha*.
a	(local)REAL for pstran DOUBLE PRECISION for pdtran Array, size (*lld_a*, LOCq(*ja*+*m*-1)). This array contains the local pieces of the distributed matrix sub(*A*).
ia, *ja*	(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.
beta	(global)REAL for pstran DOUBLE PRECISION for pdtran Specifies the scalar *beta*. When *beta* is equal to zero, then sub(*C*) need not be set on input.
c	(local)REAL for pstran DOUBLE PRECISION for pdtran Array, size (*lld_c*, LOCq(*jc*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*C*).
ic, *jc*	(global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.
descc	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c	Overwritten by the updated submatrix.

p?tranu

Transposes a distributed complex matrix.

Syntax

```
call pctranu(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

```
call pztranu(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?tranu routines transpose a complex distributed matrix. The operation is defined as

```
sub(C):=beta*sub(C) + alpha*sub(A)',
```

where:

alpha and *beta* are scalars,

sub(*C*) is an *m*-by-*n* distributed matrix, sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*m*-1).

Input Parameters

m	(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*C*), *m*≥ 0.
n	(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*C*) , *n*≥ 0.
alpha	(global)COMPLEX for pctranu DOUBLE COMPLEX for pztranu Specifies the scalar *alpha*.
a	(local)COMPLEX for pctranu DOUBLE COMPLEX for pztranu Array, size (*lld_a*, LOCq(*ja*+*m*-1)). This array contains the local pieces of the distributed matrix sub(*A*).
ia, *ja*	(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.
beta	(global)COMPLEX for pctranu DOUBLE COMPLEX for pztranu Specifies the scalar *beta*. When *beta* is equal to zero, then sub(*C*) need not be set on input.
c	(local)COMPLEX for pctranu DOUBLE COMPLEX for pztranu Array, size (*lld_c*, LOCq(*jc*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*C*).

ic, *jc*	(global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.
descc	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c	Overwritten by the updated submatrix.

p?tranc

Transposes a complex distributed matrix, conjugated.

Syntax

```
call pctranc(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pztranc(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?tranc routines transpose a complex distributed matrix. The operation is defined as

```
sub(C):=beta*sub(C) + alpha*conjg(sub(A)'),
```

where:

alpha and *beta* are scalars,

sub(*C*) is an *m*-by-*n* distributed matrix, sub(*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*m*-1).

Input Parameters

m	(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*C*), *m*≥ 0.
n	(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*C*) , *n*≥ 0.
alpha	(global)COMPLEX for pctranc DOUBLE COMPLEX for pztranc Specifies the scalar *alpha*.
a	(local)COMPLEX for pctranc DOUBLE COMPLEX for pztranc Array, size (*lld_a*, LOCq(*ja*+*m*-1)). This array contains the local pieces of the distributed matrix sub(*A*).

ia, *ja*	(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.
beta	(global)COMPLEX for pctranc DOUBLE COMPLEX for pztranc Specifies the scalar *beta*. When *beta* is equal to zero, then sub(*C*) need not be set on input.
c	(local)COMPLEX for pctranc DOUBLE COMPLEX for pztranc Array, size (*lld_c*, LOCq(*jc*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*C*).
ic, *jc*	(global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.
descc	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c	Overwritten by the updated submatrix.

p?trmm

Computes a scalar-matrix-matrix product (one matrix operand is triangular) for distributed matrices.

Syntax

```
call pstrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pdtrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pctrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pztrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
```

Include Files

- mkl_pblas.h

Description

The p?trmm routines perform a matrix-matrix operation using triangular matrices. The operation is defined as

```
sub(B) := alpha*op(sub(A))*sub(B)
```

or

```
sub(B) := alpha*sub(B)*op(sub(A))
```

where:

alpha is a scalar,

sub(*B*) is an *m*-by-*n* distributed matrix, sub(*B*)=*B*(*ib*:*ib*+*m*-1, *jb*:*jb*+*n*-1).

A is a unit, or non-unit, upper or lower triangular distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*m*-1), if *side* = 'L' or 'l', and sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), if *side* = 'R' or 'r'.

op(sub(*A*)) is one of op(sub(*A*)) = sub(*A*), or op(sub(*A*)) = sub(*A*)', or op(sub(*A*)) = conjg(sub(*A*)').

Input Parameters

side
(global) CHARACTER*1. Specifies whether op(sub(*A*)) appears on the left or right of sub(*B*) in the operation:

if *side* = 'L' or 'l', then sub(*B*) := *alpha**op(sub(*A*))*sub(*B*);

if *side* = 'R' or 'r', then sub(*B*) := *alpha**sub(*B*)*op(sub(*A*)).

uplo
(global) CHARACTER*1. Specifies whether the distributed matrix sub(*A*) is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

transa
(global) CHARACTER*1. Specifies the form of op(sub(*A*)) used in the matrix multiplication:

if *transa* = 'N' or 'n', then op(sub(*A*)) = sub(*A*);

if *transa* = 'T' or 't', then op(sub(*A*)) = sub(*A*)' ;

if *transa* = 'C' or 'c', then op(sub(*A*)) = conjg(sub(*A*)').

diag
(global) CHARACTER*1. Specifies whether the matrix sub(*A*) is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

m
(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*B*), $m \geq 0$.

n
(global) INTEGER. Specifies the number of columns of the distributed matrix sub(*B*), $n \geq 0$.

alpha
(global)REAL for pstrmm

DOUBLE PRECISION for pdtrmm

COMPLEX for pctrmm

DOUBLE COMPLEX for pztrmm

Specifies the scalar *alpha*.

When *alpha* is zero, then the array*b* need not be set before entry.

a
(local)REAL for pstrmm

DOUBLE PRECISION for pdtrmm

COMPLEX for pctrmm

DOUBLE COMPLEX for pztrmm

Array, size *lld_a* by *ka*, where *ka* is at least LOCq(1, *ja*+*m*-1) when *side* = 'L' or 'l' and is at least LOCq(1, *ja*+*n*-1) when *side* = 'R' or 'r'.

Before entry with *uplo* = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix sub(*A*) is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix sub(*A*) is not referenced .

When *diag* = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix sub(*A*) are not referenced either, but are assumed to be unity.

ia, *ja* — (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

b — (local)REAL for pstrmm

DOUBLE PRECISION for pdtrmm

COMPLEX for pctrmm

DOUBLE COMPLEX for pztrmm

Array, size (*lld_b*, LOCq(1, *jb*+*n*-1)).

Before entry, this array contains the local pieces of the distributed matrix sub(*B*).

ib, *jb* — (global) INTEGER. The row and column indices in the distributed matrix *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb — (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *B*.

Output Parameters

b — Overwritten by the transformed distributed matrix.

p?trsm

Solves a distributed matrix equation (one matrix operand is triangular).

Syntax

```
call pstrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pdtrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pctrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
```

```
call pztrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
```

Include Files

- mkl_pblas.h

Description

The p?trsm routines solve one of the following distributed matrix equations:

```
op(sub(A))*X = alpha*sub(B),
```

or

```
X*op(sub(A)) = alpha*sub(B),
```

where:

alpha is a scalar,

X and sub(*B*) are *m*-by-*n* distributed matrices, sub(*B*)=*B*(*ib*:*ib*+*m*-1, *jb*:*jb*+*n*-1);

A is a unit, or non-unit, upper or lower triangular distributed matrix, sub(*A*)=*A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*m*-1), if *side* = 'L' or 'l', and sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), if *side* = 'R' or 'r';

op(sub(*A*)) is one of op(sub(*A*)) = sub(*A*), or op(sub(*A*)) = sub(*A*)', or op(sub(*A*)) = conjg(sub(*A*)').

The distributed matrix sub(*B*) is overwritten by the solution matrix *X*.

Input Parameters

side

(global) CHARACTER*1. Specifies whether op(sub(*A*)) appears on the left or right of *X* in the equation:

if *side* = 'L' or 'l', then op(sub(*A*))*X = *alpha*sub(*B*);

if *side* = 'R' or 'r', then *X*op(sub(*A*)) = *alpha*sub(*B*).

uplo

(global) CHARACTER*1. Specifies whether the distributed matrix sub(*A*) is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

transa

(global) CHARACTER*1. Specifies the form of op(sub(*A*)) used in the matrix equation:

if *transa* = 'N' or 'n', then op(sub(*A*)) = sub(*A*);

if *transa* = 'T' or 't', then op(sub(*A*)) = sub(*A*)';

if *transa* = 'C' or 'c', then op(sub(*A*)) = conjg(sub(*A*)').

diag

(global) CHARACTER*1. Specifies whether the matrix sub(*A*) is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;

if *diag* = 'N' or 'n', then the matrix is not unit triangular.

m

(global) INTEGER. Specifies the number of rows of the distributed matrix sub(*B*), $m \geq 0$.

n (global) INTEGER. Specifies the number of columns of the distributed matrix sub(*B*), *n*≥ 0.

alpha (global)REAL for pstrsm

DOUBLE PRECISION for pdtrsm

COMPLEX for pctrsm

DOUBLE COMPLEX for pztrsm

Specifies the scalar *alpha*.

When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.

a (local)REAL for pstrsm

DOUBLE PRECISION for pdtrsm

COMPLEX for pctrsm

DOUBLE COMPLEX for pztrsm

Array, size *lld_a* by *ka*, where *ka* is at least LOCq(1, *ja*+*m*-1) when *side* = 'L' or 'l' and is at least LOCq(1, *ja*+*n*-1) when *side* = 'R' or 'r'.

Before entry with *uplo* = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix sub(*A*) is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix sub(*A*), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix sub(*A*) is not referenced .

When *diag* = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix sub(*A*) are not referenced either, but are assumed to be unity.

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

b (local)REAL for pstrsm

DOUBLE PRECISION for pdtrsm

COMPLEX for pctrsm

DOUBLE COMPLEX for pztrsm

Array, size (*lld_b*, LOCq(1, *jb*+*n*-1)).

Before entry, this array contains the local pieces of the distributed matrix sub(*B*).

ib, *jb*	(global) INTEGER. The row and column indices in the distributed matrix *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.
descb	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *B*.

Output Parameters

b	Overwritten by the solution distributed matrix *X*.

Partial Differential Equations Support

The Intel® oneAPI Math Kernel Library (oneMKL) provides tools for solving Partial Differential Equations (PDE). These tools are Trigonometric Transform interface routines (seeTrigonometric Transform Routines) and Poisson Solver (see Fast Poisson Solver Routines).

Poisson Solver is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The solver is based on the Trigonometric Transform interface, which is, in turn, based on the Intel® oneAPI Math Kernel Library (oneMKL) Fast Fourier Transform (FFT) interface (refer toFourier Transform Functions), optimized for Intel® processors.

Direct use of the Trigonometric Transform routines may be helpful to those who have already implemented their own solvers similar to the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver. As it may be hard enough to modify the original code so as to make it work with Poisson Solver, you are encouraged to use fast (staggered) sine/cosine transforms implemented in the Trigonometric Transform interface to improve performance of your solver.

Both Trigonometric Transform and Poisson Solver routines can be called from C and Fortran, although the interfaces description uses C convention. Fortran users can find routine calls specifics in Calling PDE Support Routines from Fortran.

NOTE
Intel® oneAPI Math Kernel Library (oneMKL) Trigonometric Transform and Poisson Solver routines support Fortran versions starting with Fortran 90.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Trigonometric Transform Routines

In addition to the Fast Fourier Transform (FFT) interface, described in Fast Fourier Transforms, Intel® oneAPI Math Kernel Library (oneMKL) supports theReal Discrete Trigonometric Transforms (sometimes called real-to-real Discrete Fourier Transforms) interface. In this document, the interface is referred to as TT interface. It implements a group of routines (TT routines) used to compute sine/cosine, staggered sine/cosine, and twice staggered sine/cosine transforms (referred to as staggered2 sine/cosine transforms, for brevity). The TT interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manually tuning routine parameters or just call routines with default parameter values. The current Intel® oneAPI Math Kernel Library (oneMKL) implementation of the TT interface can be used in solving partial differential equations and contains routines that are helpful for Fast Poisson and similar solvers.

To describe the Intel® oneAPI Math Kernel Library (oneMKL) TT interface, the C convention is used. Fortran users should refer toCalling PDE Support Routines from Fortran.

For the list of Trigonometric Transforms currently implemented in Intel® oneAPI Math Kernel Library (oneMKL) TT interface, seeTransforms Implemented.

If you have got used to the FFTW interface (www.fftw.org), you can call the TT interface functions through real-to-real FFTW to Intel® oneAPI Math Kernel Library (oneMKL) wrappers without changing FFTW function calls in your code (refer toFFTW to Intel® MKL Wrappers for FFTW 3.x for details). However, you are strongly encouraged to use the native TT interface for better performance. Another reason why you should use the wrappers cautiously is that TT and the real-to-real FFTW interfaces are not fully compatible and some features of the real-to-real FFTW, such as strides and multidimensional transforms, are not available through wrappers.

Trigonometric Transforms Implemented

TT routines allow computing the following transforms:

Forward sine transform

$$F(k) = \frac{2}{n}\sum_{i=1}^{n-1} f(i)\sin\frac{ki\pi}{n},\ k = 1,\ ...,\ n-1$$

Backward sine transform

$$f(i) = \sum_{k=1}^{n-1} F(k)\sin\frac{ki\pi}{n},\ i = 1,\ ...,\ n-1$$

Forward staggered sine transform

$$F(k) = \frac{1}{n}\sin\frac{(2k-1)\pi}{2}f(n) + \frac{2}{n}\sum_{i=1}^{n-1} f(i)\sin\frac{(2k-1)i\pi}{2n},\ k = 1,\ ...,\ n$$

Backward staggered sine transform

$$f(i) = \sum_{k=1}^{n} F(k)\sin\frac{(2k-1)i\pi}{2n},\ i = 1,\ ...,\ n$$

Forward staggered2 sine transform

$$F(k) = \frac{2}{n}\sum_{i=1}^{n} f(i)\sin\frac{(2k-1)(2i-1)\pi}{4n},\ k = 1,\ ...,\ n$$

Backward staggered2 sine transform

$$f(i) = \sum_{k=1}^{n} F(k)\sin\frac{(2k-1)(2i-1)\pi}{4n},\ i = 1,\ ...,\ n$$

Forward cosine transform

$$F(k) = \frac{1}{n}[f(o) + f(n)\cos k\pi] + \frac{2}{n}\sum_{i=1}^{n-1} f(i)\cos\frac{ki\pi}{n},\ k = 0,\ ...,\ n$$

Backward cosine transform

$$f(i) = \frac{1}{2}[F(o) + F(n)\cos i\pi] + \sum_{k=1}^{n-1} F(k)\cos\frac{ki\pi}{n},\ i = 0,\ ...,\ n$$

Forward staggered cosine transform

$$F(k) = \frac{1}{n}f(0) + \frac{2}{n}\sum_{i=1}^{n-1} f(i)\cos\frac{(2k+1)i\pi}{2n},\ k = 0,\ ...,\ n-1$$

Backward staggered cosine transform

$$f(i) = \sum_{k=0}^{n-1} F(k) \cos\frac{(2k+1)i\pi}{2n},\ i = 0,\ ...,\ n-1$$

Forward staggered2 cosine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^{n} f(i) \cos\frac{(2k-1)(2i-1)\pi}{4n},\ k = 1,\ ...,\ n$$

Backward staggered2 cosine transform

$$f(i) = \sum_{k=1}^{n} F(k) \cos\frac{(2k-1)(2i-1)\pi}{4n},\ i = 1,\ ...,\ n$$

NOTE
The size of the transform *n* can be any integer greater or equal to 2.

Sequence of Invoking TT Routines

Computation of a transform using TT interface is conceptually divided into four steps, each of which is performed via a dedicated routine. Table "TT Interface Routines" lists the routines and briefly describes their purpose and use.

Most TT routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "`s`" and "`d`". The wildcard "`?`" stands for either of these symbols in routine names.

TT Interface Routines

Routine	Description
`?_init_trig_transform`	Initializes basic data structures of Trigonometric Transforms.
`?_commit_trig_transform`	Checks consistency and correctness of user-defined data and creates a data structure to be used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface[1].
`?_forward_trig_transform` `?_backward_trig_transform`	Computes a forward/backward Trigonometric Transform of a specified type using the appropriate formula (see Transforms Implemented).
`free_trig_transform`	Releases the memory used by a data structure needed for calling FFT interface[1].

[1]TT routines call Intel® oneAPI Math Kernel Library (oneMKL) FFT interface for better performance.

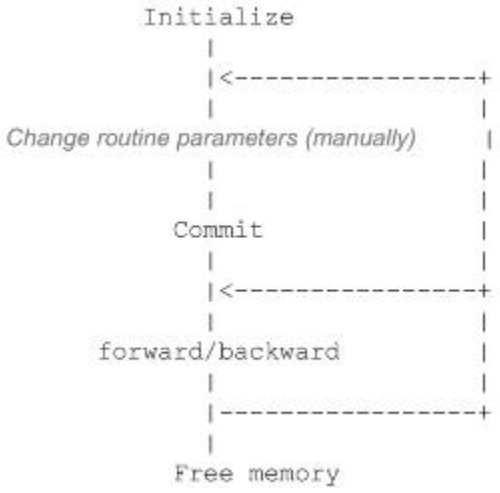
To find a transformed vector for a particular input vector only once, the Intel® oneAPI Math Kernel Library (oneMKL) TT interface routines are normally invoked in the order in which they are listed inTable "TT Interface Routines".

NOTE
Though the order of invoking TT routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in Figure "Typical Order of Invoking TT Interface Routines" indicates the typical order in which TT interface routines can be invoked in a general case (prefixes and suffixes in routine names are omitted).

__border__top

Typical Order of Invoking TT Interface Routines



A general scheme of using TT routines for double-precision computations is shown below. A similar scheme holds for single-precision computations with the only difference in the initial letter of routine names.

```
...
    d_init_trig_transform(&n, &tt_type, ipar, dpar, &ir);
/* Change parameters in ipar if necessary. */
/* Note that the result of the Transform will be in f. If you want to preserve the data stored
in f,
save it to another location before the function call below */
    d_commit_trig_transform(f, &handle, ipar, dpar, &ir);
    d_forward_trig_transform(f, &handle, ipar, dpar, &ir);
    d_backward_trig_transform(f, &handle, ipar, dpar, &ir);
    free_trig_transform(&handle, ipar, &ir);
/* here the user may clean the memory used by f, dpar, ipar */
...
```

You can find examples of code that uses TT interface routines to solve one-dimensional Helmholtz problem in the `examples\pdettf\source` folderin your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Trigonometric Transform Interface Description

All types in this documentation are either standard C types `float` and `double` or `MKL_INT` integer type. Fortran users can call the routines with `REAL` and `DOUBLE PRECISION` types of floating-point values and `INTEGER` or `INTEGER`*8 integer type depending on the programming interface (LP64 or ILP64). To better understand usage of the types, see examples in the `examples\pdettf\source` folderin your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Routine Options

All TT routines use parameters to pass various options to one another. These parameters are arrays *ipar*, *dpar* and *spar*. Values for these parameters should be specified very carefully (see Common Parameters). You can change these values during computations to meet your needs.

WARNING
To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.

User Data Arrays

TT routines take arrays of user data as input. For example, user arrays are passed to the routine `d_forward_trig_transform`to compute a forward Trigonometric Transform. To minimize storage requirements and improve the overall run-time efficiency, Intel® oneAPI Math Kernel Library (oneMKL) TT routines do not make copies of user input arrays.

> **NOTE**
> If you need a copy of your input data arrays, you must save them yourself.

For better performance, align your data arrays as recommended in the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* (search the document for coding techniques to improve performance).

TT Routines

The section gives detailed description of TT routines, their syntax, parameters and values they return. Double-precision and single-precision versions of the same routine are described together.

TT routines call Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (described inFFT Functions), which enhances performance of the routines.

?_init_trig_transform

Initializes basic data structures of a Trigonometric Transform.

Syntax

```
void d_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], double dpar[],
MKL_INT *stat);
```

```
void s_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], float spar[],
MKL_INT *stat);
```

Include Files

- `mkl_trig_transforms.f90`

Input Parameters

n	`MKL_INT*`. Contains the size of the problem, which should be a positive integer greater than 1. Note that data vector of the transform, which other TT routines will use, must have size *n*+1 for all but staggered2 transforms. Staggered2 transforms require the vector of size *n*.
tt_type	`MKL_INT*`. Contains the type of transform to compute, defined via a set of named constants. The following constants are available in the current implementation of TT interface: `MKL_SINE_TRANSFORM`, `MKL_STAGGERED_SINE_TRANSFORM`, `MKL_STAGGERED2_SINE_TRANSFORM`; `MKL_COSINE_TRANSFORM`, `MKL_STAGGERED_COSINE_TRANSFORM`, `MKL_STAGGERED2_COSINE_TRANSFORM`.

Output Parameters

ipar	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
dpar	double array of size 5*n*/2+2. Contains double-precision data needed for Trigonometric Transform computations.
spar	float array of size 5*n*/2+2. Contains single-precision data needed for Trigonometric Transform computations.
stat	MKL_INT*. Contains the routine completion status, which is also written to *ipar[6]*. The status should be 0 to proceed to other TT routines.

Description

The ?_init_trig_transform routine initializes basic data structures for Trigonometric Transforms of appropriate precision. After a call to ?_init_trig_transform, all subsequently invoked TT routines use values of *ipar* and *dpar* (*spar*) array parameters returned by ?_init_trig_transform. The routine initializes the entire array *ipar*. In the *dpar* or *spar* array, ?_init_trig_transform initializes elements that do not depend upon the type of transform. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to Common Parameters. You can skip a call to the initialization routine in your code. For more information, see Caveat on Parameter Modifications.

Return Values

stat= 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this *stat* value.
stat= -99999	The routine failed to complete the task.

?_commit_trig_transform

Checks consistency and correctness of user's data as well as initializes certain data structures required to perform the Trigonometric Transform.

Syntax

```
void d_commit_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
```

```
void s_commit_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- mkl_trig_transforms.f90

Input Parameters

f	double for d_commit_trig_transform, float for s_commit_trig_transform,

array of size *n* for staggered2 transforms and of size *n*+1 for all other transforms, where *n* is the size of the problem. Contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors:

- *f*[0] and *f*[*n*] for sine transforms
- *f*[*n*] for staggered cosine transforms
- *f*[0] for staggered sine transforms.

Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. These restrictions meet the requirements of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver, which the TT interface is primarily designed for (for details, seeFast Poisson Solver Routines).

ipar — `MKL_INT` array of size 128. Contains integer data needed for Trigonometric Transform computations.

dpar — `double` array of size 5*n*/2+2. Contains double-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.

spar — `float` array of size 5*n*/2+2. Contains single-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.

Output Parameters

handle — `DFTI_DESCRIPTOR_HANDLE*`. The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer toFFT Functions).

ipar — Contains integer data needed for Trigonometric Transform computations. On output, *ipar[6]* is updated with the *stat* value.

dpar — Contains double-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.

spar — Contains single-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.

stat — `MKL_INT*`. Contains the routine completion status, which is also written to *ipar[6]*.

Description

The routine `?_commit_trig_transform` checks consistency and correctness of the parameters to be passed to the transform routines `?_forward_trig_transform` and/or `?_backward_trig_transform`. The routine also initializes the following data structures: *handle*, *dpar* in case of `d_commit_trig_transform`, and *spar* in case of `s_commit_trig_transform`. The `?_commit_trig_transform` routine initializes only those elements of *dpar* or *spar* that depend upon the type of transform, defined in the `?_init_trig_transform` routine and passed to `?_commit_trig_transform` with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to Common Parameters. The routine performs only a basic check for correctness and consistency

of the parameters. If you are going to modify parameters of TT routines, see Caveat on Parameter Modifications. Unlike `?_init_trig_transform`, you must call the `?_commit_trig_transform` routine in your code.

Return Values

stat= 11	The routine produced some warnings and made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning *ipar[6]*=0 if you are sure that the parameters are correct.
stat= 10	The routine made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning *ipar[6]*=0 if you are sure that the parameters are correct.
stat= 1	The routine produced some warnings. You may proceed with computations by assigning *ipar[6]*=0 if you are sure that the parameters are correct.
stat= 0	The routine completed the task normally.
stat= -100	The routine stopped for any of the following reasons: • An error in the user's data was encountered. • Data in *ipar*, *dpar* or *spar* parameters became incorrect and/or inconsistent as a result of modifications.
stat= -1000	The routine stopped because of an FFT interface error.
stat= -10000	The routine stopped because the initialization failed to complete or the parameter *ipar[0]* was altered by mistake.

NOTE
Although positive values of *stat* usually indicate minor problems with the input data and Trigonometric Transform computations can be continued, you are highly recommended to investigate the problem first and achieve *stat*=0.

?_forward_trig_transform

Computes the forward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_forward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
```

```
void s_forward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- `mkl_trig_transforms.f90`

Input Parameters

f	`double` for `d_forward_trig_transform`, `float` for `s_forward_trig_transform`, array of size *n* for staggered2 transforms and of size *n*+1 for all other transforms, where *n* is the size of the problem. On input, contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors: • *f*[0] and *f*[*n*] for sine transforms • *f*[*n*] for staggered cosine transforms • *f*[0] for staggered sine transforms. Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver, which the TT interface is primarily designed for (for details, seeFast Poisson Solver Routines).
handle	`DFTI_DESCRIPTOR_HANDLE*`. The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, seeFFT Functions).
ipar	`MKL_INT` array of size 128. Contains integer data needed for Trigonometric Transform computations.
dpar	`double` array of size 5*n*/2+2. Contains double-precision data needed for Trigonometric Transform computations.
spar	`float` array of size 5*n*/2+2. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

f	Contains the transformed vector on output.
ipar	Contains integer data needed for Trigonometric Transform computations. On output, *ipar[6]* is updated with the *stat* value.
stat	`MKL_INT*`. Contains the routine completion status, which is also written to *ipar[6]*.

Description

The routine computes the forward Trigonometric Transform of type defined in the `?_init_trig_transform` routine and passed to `?_forward_trig_transform` with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. The other data that facilitates the computation is created by `?_commit_trig_transform` and supplied in *dpar* or *spar*. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to Common Parameters. The routine has a commit step, which calls the `?_commit_trig_transform` routine. The transform is computed according to formulas given in Transforms Implemented. The routine replaces the input vector *f* with the transformed vector.

NOTE
If you need a copy of the data vector *f* to be transformed, make the copy before calling the ?_forward_trig_transform routine.

Return Values

stat= 0	The routine completed the task normally.
stat= -100	The routine stopped for any of the following reasons: • An error in the user's data was encountered. • Data in *ipar*, *dpar* or *spar* parameters became incorrect and/or inconsistent as a result of modifications.
stat= -1000	The routine stopped because of an FFT interface error.
stat= -10000	The routine stopped because its commit step failed to complete or the parameter *ipar[0]* was altered by mistake.

?_backward_trig_transform

Computes the backward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_backward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
```

```
void s_backward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- mkl_trig_transforms.f90

Input Parameters

f	double for d_backward_trig_transform, float for s_backward_trig_transform, array of size *n* for staggered2 transforms and of size *n*+1 for all other transforms, where *n* is the size of the problem. On input, contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors: • *f*[0] and *f*[*n*] for sine transforms • *f*[*n*] for staggered cosine transforms • *f*[0] for staggered sine transforms. Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver, which the TT interface is primarily designed for (for details, seeFast Poisson Solver Routines).

handle	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, seeFFT Functions).
ipar	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
dpar	double array of size 5*n*/2+2. Contains double-precision data needed for Trigonometric Transform computations.
spar	float array of size 5*n*/2+2. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

f	Contains the transformed vector on output.
ipar	Contains integer data needed for Trigonometric Transform computations. On output, *ipar[6]* is updated with the *stat* value.
stat	MKL_INT*. Contains the routine completion status, which is also written to *ipar[6]*.

Description

The routine computes the backward Trigonometric Transform of type defined in the ?_init_trig_transform routine and passed to ?_backward_trig_transform with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called ?_init_trig_transform routine. The other data that facilitates the computation is created by ?_commit_trig_transform and supplied in *dpar* or *spar*. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to Common Parameters. The routine has a commit step, which calls the ?_commit_trig_transform routine. The transform is computed according to formulas given in Transforms Implemented. The routine replaces the input vector *f* with the transformed vector.

NOTE
If you need a copy of the data vector *f* to be transformed, make the copy before calling the ?_backward_trig_transform routine.

Return Values

stat= 0	The routine completed the task normally.
stat= -100	The routine stopped for any of the following reasons: • An error in the user's data was encountered. • Data in *ipar*, *dpar* or *spar* parameters became incorrect and/or inconsistent as a result of modifications.
stat= -1000	The routine stopped because of an FFT interface error.
stat= -10000	The routine stopped because its commit step failed to complete or the parameter *ipar[0]* was altered by mistake.

free_trig_transform

Cleans the memory allocated for the data structure used by the FFT interface.

Syntax

```
void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], MKL_INT *stat);
```

Include Files

- mkl_trig_transforms.f90

Input Parameters

ipar	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
handle	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer toFFT Functions).

Output Parameters

handle	The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface. Memory allocated for the structure is released on output.
ipar	Contains integer data needed for Trigonometric Transform computations. On output, *ipar[6]* is updated with the *stat* value.
stat	MKL_INT*. Contains the routine completion status, which is also written to *ipar[6]*.

Description

The free_trig_transform routine cleans the memory used by the *handle*structure, needed for Intel® oneAPI Math Kernel Library (oneMKL) FFT functions. To release the memory allocated for other parameters, include cleaning of the memory in your code.

Return Values

stat= 0	The routine completed the task normally.
stat= -1000	The routine stopped because of an FFT interface error.
stat= -99999	The routine failed to complete the task.

Common Parameters of the Trigonometric Transforms

This section provides description of array parameters that hold TT routine options: *ipar*, *dpar* and *spar*.

NOTE
Initial values are assigned to the array parameters by the appropriate ?_init_trig_transform and ?_commit_trig_transform routines.

ipar	MKL_INT array of size 128, holds integer data needed for Trigonometric Transform computations. Its elements are described in Table "Elements of the ipar Array":

Elements of the ipar Array

Index	Description
0	Contains the size of the problem to solve. The ?_init_trig_transform routine sets *ipar[0]*=*n*, and all subsequently called TT routines use *ipar[0]* as the size of the transform.
1	Contains error messaging options: • *ipar[1]*=-1 indicates that all error messages will be printed to the file MKL_Trig_Transforms_log.txt in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • *ipar[1]*=0 indicates that no error messages will be printed. • *ipar[1]*=1 (default) indicates that all error messages will be printed to the preconnected default output device (usually, screen). In case of errors, each TT routine assigns a non-zero value to *stat* regardless of the *ipar[1]* setting.
2	Contains warning messaging options: • *ipar[2]*=-1 indicates that all warning messages will be printed to the file MKL_Trig_Transforms_log.txt in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • *ipar[2]*=0 indicates that no warning messages will be printed. • *ipar[2]*=1 (default) indicates that all warning messages will be printed to the preconnected default output device (usually, screen). In case of warnings, the *stat* parameter will acquire a non-zero value regardless of the *ipar[2]* setting.
3 through 4	Reserved for future use.
5	Contains the type of the transform. The ?_init_trig_transform routine sets *ipar[5]*=*tt_type*, and all subsequently called TT routines use *ipar[5]* as the type of the transform.
6	Contains the *stat* value returned by the last completed TT routine. Used to check that the previous call to a TT routine completed with *stat*=0.
7	Informs the ?_commit_trig_transform routines whether to initialize data structures *dpar* (*spar*) and *handle*. *ipar[7]*=0 indicates that the routine should skip the initialization and only check correctness and consistency of the parameters. Otherwise, the routine initializes the data structures. The default value is 1. The possibility to check correctness and consistency of input data without initializing data structures *dpar*, *spar* and *handle* enables avoiding performance losses in a repeated use of the same transform for different data vectors. Note that you can benefit from the opportunity that *ipar[7]* gives only if you are sure to have supplied proper tolerance value in the *dpar* or *spar* array. Otherwise, avoid tuning this parameter.
8	Contains message style options for TT routines. If *ipar[8]*=0 then TT routines print all error and warning messages in Fortran-style notations. The default value is 1.

Index	Description
	When specifying message style options, be aware that by default, numbering of elements in Fortran arrays starts at 1. The use of *ipar[8]* enables you to view messages in a more convenient style.
9	Specifies the number of OpenMP threads to run TT routines in the OpenMP environment of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver. The default value is 1. You are highly recommended not to alter this value. See alsoCaveat on Parameter Modifications.
10	Specifies the mode of compatibility with FFTW. The default value is 0. Set the value to 1 to invoke compatibility with FFTW. In the latter case, results will not be normalized, because FFTW does not do this. It is highly recommended not to alter this value, but rather use real-to-real FFTW to MKL wrappers, described in FFTW to Intel® MKL Wrappers for FFTW 3.x. See also Caveat on Parameter Modifications.
11 through 127	Reserved for future use.

NOTE
While you can declare the *ipar* array as `MKL_INT ipar[11]`, for future compatibility you should declare *ipar* as `MKL_INT ipar[128]`.

Arrays *dpar* and *spar* are the same except in the data precision:

dpar	`double` array of size 5*n*/2+2, holds data needed for double-precision routines to perform TT computations. This array is initialized in the `d_init_trig_transform` and `d_commit_trig_transform` routines.
spar	`float` array of size 5*n*/2+2, holds data needed for single-precision routines to perform TT computations. This array is initialized in the `s_init_trig_transform` and `s_commit_trig_transform` routines.

As *dpar* and *spar* have similar elements in respective positions, the elements are described together in Table "Elements of the dpar and spar Arrays":

Elements of the dpar and spar Arrays

Index	Description
0	Contains the first absolute tolerance used by the appropriate `?_commit_trig_transform` routine. For a staggered cosine or a sine transform, *f[n]* should be equal to 0.0 and for a staggered sine or a sine transform, *f[0]* should be equal to 0.0. The `?_commit_trig_transform` routine checks whether absolute values of these parameters are below *dpar[0]*n* or *spar[0]*n*, depending on the routine precision. To suppress warnings resulting from tolerance checks, set *dpar[0]* or *spar[0]* to a sufficiently large number.
1	Reserved for future use.
2 through 5*n*/2+1	Contain tabulated values of trigonometric functions. Contents of the elements depend upon the type of transform *tt_type*, set up in the `?_commit_trig_transform` routine: • If *tt_type*=`MKL_SINE_TRANSFORM`, the transform uses only the first *n*/2 array elements, which contain tabulated sine values. • If *tt_type*=`MKL_STAGGERED_SINE_TRANSFORM`, the transform uses only the first 3*n*/2 array elements, which contain tabulated sine and cosine values. • If *tt_type*=`MKL_STAGGERED2_SINE_TRANSFORM`, the transform uses all the 5*n*/2 array elements, which contain tabulated sine and cosine values.

Index	Description
	• If *tt_type*=MKL_COSINE_TRANSFORM, the transform uses only the first *n* array elements, which contain tabulated cosine values. • If *tt_type*=MKL_STAGGERED_COSINE_TRANSFORM, the transform uses only the first 3*n*/2 elements, which contain tabulated sine and cosine values. • If *tt_type*=MKL_STAGGERED2_COSINE_TRANSFORM, the transform uses all the 5*n*/2 elements, which contain tabulated sine and cosine values.

NOTE
To save memory, you can define the array size depending upon the type of transform.

Caveat on Parameter Modifications

Flexibility of the TT interface enables you to skip a call to the ?_init_trig_transform routine and to initialize the basic data structures explicitly in your code. You may also need to modify the contents of *ipar*, *dpar* and *spar* arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or wrong computation. You can perform a basic check for correctness and consistency of parameters by calling the ?_commit_trig_transform routine; however, this does not ensure the correct result of a transform but only reduces the chance of errors or wrong results.

NOTE
To supply correct and consistent parameters to TT routines, you should have considerable experience in using the TT interface and good understanding of elements that the *ipar*, *spar* and *dpar* arrays contain and dependencies between values of these elements.

However, in rare occurrences, even advanced users might fail to compute a transform using TT routines after the parameter modifications. In cases like these, refer for technical support at http://www.intel.com/software/products/support/ .

WARNING
The only way that ensures proper computation of the Trigonometric Transforms is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of *ipar*, *dpar* and *spar* arrays unless a strong need arises.

Trigonometric Transform Implementation Details

Several aspects of the Intel® oneAPI Math Kernel Library (oneMKL) TT interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, Intel® oneAPI Math Kernel Library (oneMKL) provides you with the TT language-specific header file to include in your code:

- mkl_trig_transforms.f90, to be used together with mkl_dfti.f90.

NOTE

- Intel® oneAPI Math Kernel Library (oneMKL) TT interface supports Fortran versions starting with Fortran 90.
- Use of the Intel® oneAPI Math Kernel Library (oneMKL) TT software without including the above language-specific header files is not supported.

Header File

The header file below defines the following function prototypes:

```
SUBROUTINE D_INIT_TRIG_TRANSFORM(n, tt_type, ipar, dpar, stat)
    INTEGER, INTENT(IN) :: n, tt_type
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_INIT_TRIG_TRANSFORM

SUBROUTINE D_COMMIT_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
    REAL(8), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_COMMIT_TRIG_TRANSFORM

SUBROUTINE D_FORWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
    REAL(8), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_FORWARD_TRIG_TRANSFORM

SUBROUTINE D_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
    REAL(8), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_BACKWARD_TRIG_TRANSFORM

SUBROUTINE S_INIT_TRIG_TRANSFORM(n, tt_type, ipar, spar, stat)
    INTEGER, INTENT(IN) :: n, tt_type
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_INIT_TRIG_TRANSFORM

SUBROUTINE S_COMMIT_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
    REAL(4), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_COMMIT_TRIG_TRANSFORM

SUBROUTINE S_FORWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
```

```
    REAL(4), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_FORWARD_TRIG_TRANSFORM

SUBROUTINE S_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
    REAL(4), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_BACKWARD_TRIG_TRANSFORM

SUBROUTINE FREE_TRIG_TRANSFORM(handle, ipar, stat)
    INTEGER, INTENT(INOUT) :: ipar(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE FREE_TRIG_TRANSFORM
```

Fortran specifics of the TT routines usage are similar for all Intel® oneAPI Math Kernel Library (oneMKL) PDE support tools and described inCalling PDE Support Routines from Fortran.

Fast Poisson Solver Routines

In addition to the Real Discrete Trigonometric Transforms (TT) interface (refer to Trigonometric Transform Routines), Intel® oneAPI Math Kernel Library (oneMKL) supports thethe Poisson Solver interface. This interface implements a group of routines (Poisson Solver routines) used to compute a solution of Laplace, Poisson, and Helmholtz problems of a special kind using discrete Fourier transforms. Laplace and Poisson problems are special cases of a more general Helmholtz problem. The problems that are solved by the Poisson Solver interface are defined more exactly in Poisson Solver Implementation. The Poisson Solver interface provides much flexibility of use: you can call routines with the default parameter values or adjust routines to your particular needs by manually tuning routine parameters. You can adjust the style of error and warning messages to a Fortrannotation by setting up a dedicated parameter. This adds convenience to debugging, because you can read information in the way that is natural for your code. The Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface currently contains only routines that implement the following solvers:

- Fast Laplace, Poisson and Helmholtz solvers in a Cartesian coordinate system
- Fast Poisson and Helmholtz solvers in a spherical coordinate system.

To describe the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface, the C convention is used. Fortran usage specifics can be found inCalling PDE Support Routines from Fortran.

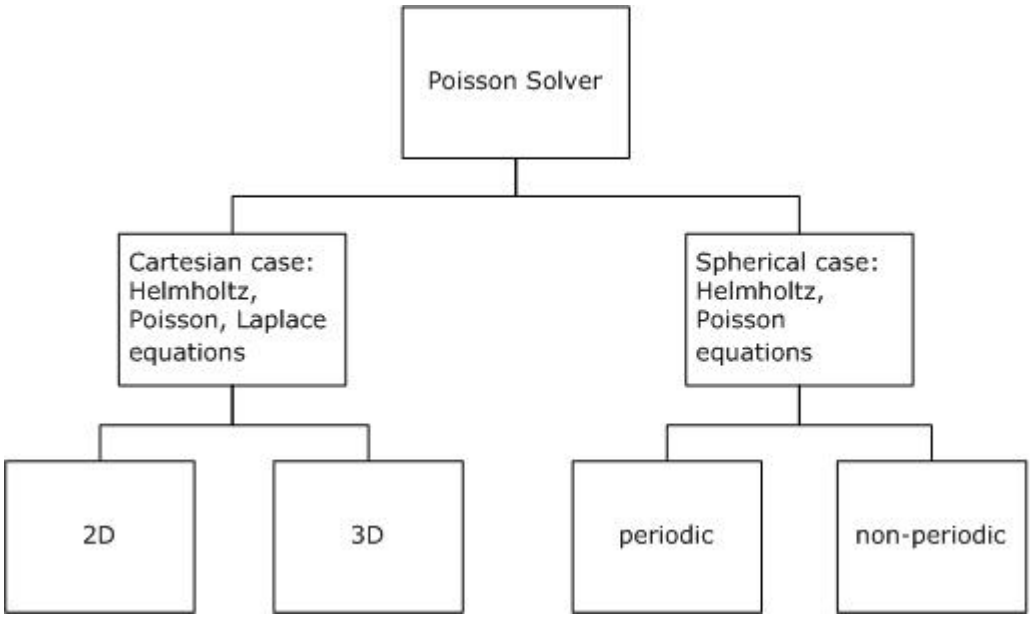
NOTE
Fortran users should keep in mind that array indices in Fortran start at 1 instead of 0, as they do in C.

Poisson Solver Implementation

Poisson Solver routines enable approximate solving of certain two-dimensional and three-dimensional problems. Figure "Structure of the Poisson Solver" shows the general structure of the Poisson Solver.

__border__top

Structure of the Poisson Solver



NOTE
Although in the Cartesian case, both periodic and non-periodic solvers are also supported, they use the same interfaces.

Sections below provide details of the problems that can be solved using Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver.

Two-Dimensional Problems

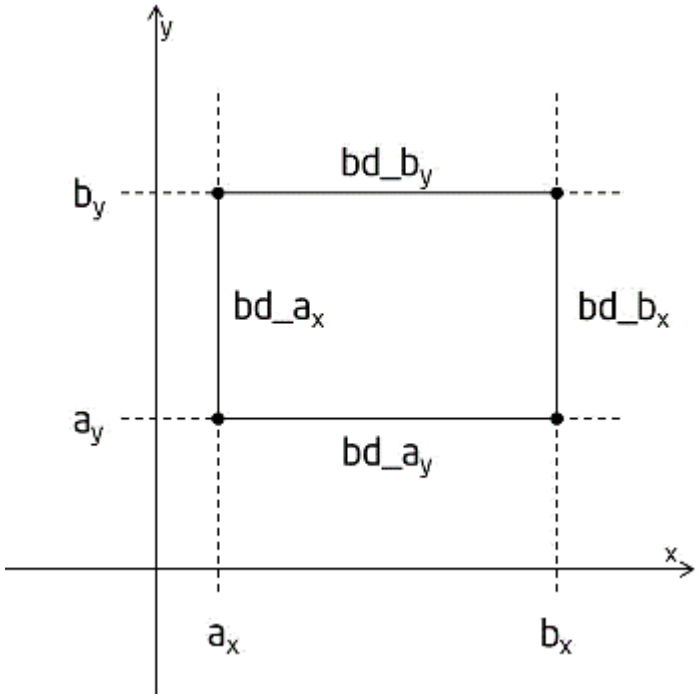
Notational Conventions

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_x < x < b_x$, $a_y < y < b_y$ on a Cartesian plane:

$bd\_a_x = \{x = a_x, a_y \le y \le b_y\}$, $bd\_b_x = \{x = b_x, a_y \le y \le b_y\}$

$bd\_a_y = \{a_x \le x \le b_x, y = a_y\}$, $bd\_b_y = \{a_x \le x \le b_x, y = b_y\}$.

The following figure shows these boundaries:



The wildcard "+" may stand for any of the symbols a_x, b_x, a_y, b_y, so *bd_+* denotes any of the above boundaries.

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_\varphi < \varphi < b_\varphi$, $a_\theta < \theta < b_\theta$ on a sphere $0 \le \varphi \le 2\pi$, $0 \le \theta \le \pi$:

bd_a$_\varphi$ = {$\varphi = a_\varphi$, $a_\theta \le \theta \le b_\theta$}, *bd_b*$_\varphi$ = {$\varphi = b_\varphi$, $a_\theta \le \theta \le b_\theta$},

bd_a$_\theta$ = {$a_\varphi \le \varphi \le b_\varphi$, $\theta = a_\theta$}, *bd_b*$_\theta$ = {$a_\varphi \le \varphi \le b_\varphi$, $\theta = b_\theta$}.

The wildcard "~" may stand for any of the symbols a_φ, b_φ, a_θ, b_θ, so *bd_~* denotes any of the above boundaries.

Two-dimensional Helmholtz problem on a Cartesian plane

The two-dimensional (2D) Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + qu = f(x,y),\ q = const \ge 0$$

in a rectangle, that is, a rectangular domain $a_x < x < b_x$, $a_y < y < b_y$, with one of the following boundary conditions on each boundary *bd_+*:

- The Dirichlet boundary condition

  $$u(x,y) = G(x,y)$$
- The Neumann boundary condition

  $$\frac{\partial u}{\partial n}(x,y) = g(x,y)$$

  where

  $n = -x$ on *bd_a*$_x$, $n = x$ on *bd_b*$_x$,

  $n = -y$ on *bd_a*$_y$, $n = y$ on *bd_b*$_y$.
- Periodic boundary conditions

  $$u(a_x,y) = u(b_x,y),\ \frac{\partial}{\partial x}u(a_x,y) = \frac{\partial}{\partial x}u(b_x,y),$$

$$u(x, a_y) = u(x, b_y),\ \frac{\partial}{\partial y}u(x, a_y) = \frac{\partial}{\partial y}u(x, b_y).$$

Two-dimensional Poisson problem on a Cartesian plane

The Poisson problem is a special case of the Helmholtz problem, when *q*=0. The 2D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

in a rectangle $a_x < x < b_x$, $a_y < y < b_y$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary *bd_+*. In case of a problem with the Neumann boundary condition on the entire boundary, you can find the solution of the problem only up to a constant. In this case, the Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

Two-dimensional (2D) Laplace problem on a Cartesian plane

The Laplace problem is a special case of the Helmholtz problem, when *q*=0 and *f(x, y)*=0. The 2D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

in a rectangle $a_x < x < b_x$, $a_y < y < b_y$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary *bd_+*.

Helmholtz problem on a sphere

The Helmholtz problem on a sphere is to find an approximate solution of the Helmholtz equation

$$-\Delta_s u + qu = f,\ q = const \geq 0,$$

$$\Delta_s = \frac{1}{\sin^2\theta}\frac{\partial^2}{\partial\phi^2} + \frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right)$$

in a domain bounded by angles $a_\varphi \leq \varphi \leq b_\varphi$, $a_\theta \leq \theta \leq b_\theta$ (spherical rectangle), with boundary conditions for particular domains listed in Table "Details of Helmholtz Problem on a Sphere".

Details of Helmholtz Problem on a Sphere

Domain on a sphere	Boundary condition	Periodic/non-periodic case
Rectangular, that is, $b_\varphi - a_\varphi < 2\pi$ and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on each boundary *bd_~*	*non-periodic*
Where $a_\varphi = 0$, $b_\varphi = 2\pi$, and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on the boundaries *bd_*a_θ and *bd_*b_θ	*periodic*
Entire sphere, that is, $a_\varphi = 0$, $b_\varphi = 2\pi$, $a_\theta = 0$, and $b_\theta = \pi$	Boundary condition $\left(\sin\theta\frac{\partial u}{\partial\theta}\right) = 0^{\theta \to 0}_{\theta \to \pi}$ at the poles	*periodic*

Poisson problem on a sphere

The Poisson problem is a special case of the Helmholtz problem, when *q*=0. The Poisson problem on a sphere is to find an approximate solution of the Poisson equation

$$-\Delta_s u = f,\ \Delta_s = \frac{1}{\sin^2\theta}\frac{\partial^2}{\partial\phi^2} + \frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right)$$

in a spherical rectangle $a_\varphi \le \varphi \le b_\varphi$, $a_\theta \le \theta \le b_\theta$ in cases listed in Table "Details of Helmholtz Problem on a Sphere". The solution to the Poisson problem on the entire sphere can be found up to a constant only. In this case, Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

Approximation of 2D problems

To find an approximate solution for any of the 2D problems, in the rectangular domain a uniform mesh can be defined for the Cartesian case as:

$$\left\{x_i = a_x + ih_x, y_j = a_y + jh_y\right\},$$

$$i = 0, ..., n_x,\ j = 0, ..., n_y,\ h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}$$

and for the spherical case as:

$$\left\{\phi_i = a_\phi + ih_\phi, \theta_j = a_\theta + jh_\theta\right\},$$

$$i = 0, ..., n_\phi,\ j = 0, ..., n_\theta,\ h_\phi = \frac{b_\phi - a_\phi}{n_\phi}, h_\theta = \frac{b_\theta - a_\theta}{n_\theta}.$$

The Poisson Solver uses the standard five-point finite difference approximation on this mesh to compute the approximation to the solution:

- In the Cartesian case, the values of the approximate solution will be computed in the mesh points (x_i , y_j) provided that you can supply the values of the right-hand side *f(x, y)* in these points and the values of the appropriate boundary functions *G(x, y)* and/or *g(x,y)* in the mesh points laying on the boundary of the rectangular domain.
- In the spherical case, the values of the approximate solution will be computed in the mesh points (φ_i , θ_j) provided that you can supply the values of the right-hand side $f(\varphi, \theta)$ in these points.

> **NOTE**
> The number of mesh intervals n_φ in the φ direction of a spherical mesh must be even in the periodic case. The Poisson Solver does not support spherical meshes that do not meet this condition.

Three-Dimensional Problems

Notational Conventions

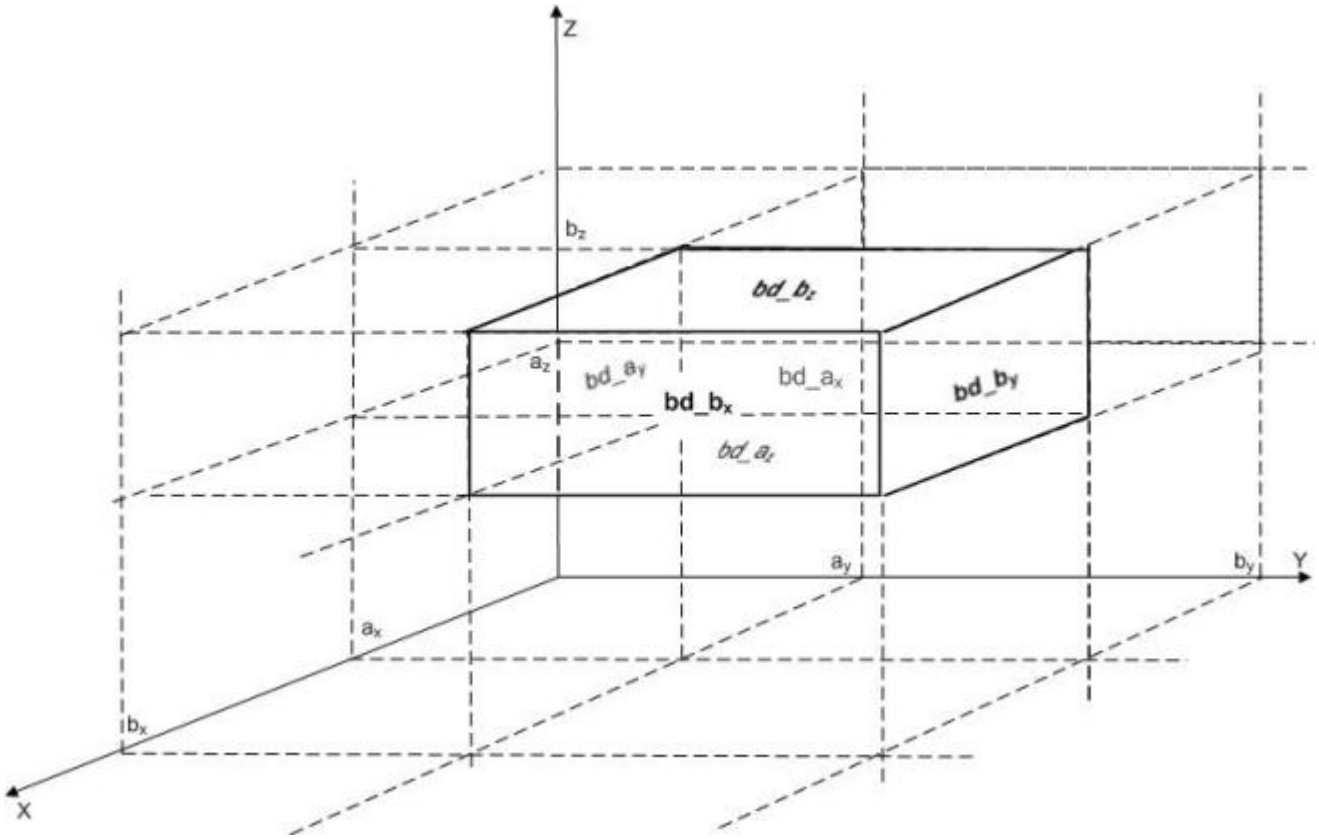
The Poisson Solver interface description uses the following notation for boundaries of a parallelepiped domain $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$:

$bd\_a_x = \{x = a_x, a_y \le y \le b_y, a_z \le z \le b_z\}$, $bd\_b_x = \{x = b_x, a_y \le y \le b_y, a_z \le z \le b_z\}$,

$bd\_a_y = \{a_x \le x \le b_x, y = a_y, a_z \le z \le b_z\}$, $bd\_b_y = \{a_x \le x \le b_x, y = b_y, a_z \le z \le b_z\}$,

$bd\_a_z = \{a_x \le x \le b_x, a_y \le y \le b_y, z = a_z\}$, $bd\_b_x = \{a_x \le x \le b_x, a_y \le y \le b_y, z = b_z\}$.

The following figure shows these boundaries:



The wildcard "+" may stand for any of the symbols a_x, b_x, a_y, b_y, a_z, b_z, so *bd_+* denotes any of the above boundaries.

Three-dimensional (3D) Helmholtz problem

The 3D Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} + qu = f(x,y,z),\ q = const \geq 0$$

in a parallelepiped, that is, a parallelepiped domain $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$, with one of the following boundary conditions on each boundary *bd_+*:

- The Dirichlet boundary condition

  $u(x,y,z) = G(x,y,z)$
- The Neumann boundary condition

  $\frac{\partial u}{\partial n}(x,y,z) = g(x,y,z)$

  where

  $n = -x$ on bd_a_x, $n = x$ on bd_b_x,

  $n = -y$ on bd_a_y, $n = y$ on bd_b_y,

  $n = -z$ on bd_a_z, $n = z$ on bd_b_z.
- Periodic boundary conditions

  $u(a_x,y,z) = u(b_x,y,z),\ \frac{\partial}{\partial x}u(a_x,y,z) = \frac{\partial}{\partial x}u(b_x,y,z),$

  $u(x,a_y,z) = u(x,\ b_y,z),\ \frac{\partial}{\partial y}u(x,a_y,z) = \frac{\partial}{\partial y}u(x,b_y,z),$

$$u(x, y, a_z) = u(x, y, b_z), \frac{\partial}{\partial z}u(x, y, a_z) = \frac{\partial}{\partial z}u(x, y, b_z).$$

Three-dimensional (3D) Poisson problem

The Poisson problem is a special case of the Helmholtz problem, when *q*=0. The 3D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f(x, y, z)$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary *bd_*+.

Three-dimensional (3D) Laplace problem

The Laplace problem is a special case of the Helmholtz problem, when *q*=0 and *f(x, y, z)*=0. The 3D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary *bd_*+.

Approximation of 3D problems

To find an approximate solution for each of the 3D problems, a uniform mesh can be defined in the parallelepiped domain as:

$$\left\{x_i = a_x + ih_x, y_j = a_y + jh_y, z_k = a_z + jh_z\right\},$$

where

$$i = 0, ..., n_x, \; j = 0, ..., n_y, \; k = 0, ..., n_z,$$

$$h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}, h_z = \frac{b_z - a_z}{n_z}.$$

The Poisson Solver uses the standard seven-point finite difference approximation on this mesh to compute the approximation to the solution. The values of the approximate solution will be computed in the mesh points (x_i, y_j, z_k), provided that you can supply the values of the right-hand side *f(x, y, z)* in these points and the values of the appropriate boundary functions *G(x, y, z)* and/or *g(x, y, z)* in the mesh points laying on the boundary of the parallelepiped domain.

Sequence of Invoking Poisson Solver Routines

NOTE
This description always shows the solution process for the Helmholtz problem, because Fast Poisson Solvers and Fast Laplace Solvers are special cases of Fast Helmholtz Solvers (see Poisson Solver Implementation).

The Poisson Solver interface enables you to compute a solution of the Helmholtz problem in four steps. Each step is performed by a dedicated routine. Table "Poisson Solver Interface Routines" lists the routines and briefly describes their purpose.

Most Poisson Solver routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names. The routines for the Cartesian coordinate system have 2D and 3D versions. Their names end respectively in "2D" and "3D". The routines for spherical coordinate system have periodic and non-periodic versions. Their names end respectively in "p" and "np".

Poisson Solver Interface Routines

Routine	Description
?_init_Helmholtz_2D/?_init_Helmholtz_3D/ ?_init_sph_p/?_init_sph_np	Initializes basic data structures for Fast Helmholtz Solver in the 2D/3D/periodic/ non-periodic case, respectively.
?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/ ?_commit_sph_p/?_commit_sph_np	Checks consistency and correctness of input data and initializes data structures for the solver, including those used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface[1].
?_Helmholtz_2D/?_Helmholtz_3D/?_sph_p/?_sph_np	Computes an approximate solution of the 2D/3D/periodic/non-periodic Helmholtz problem (see Poisson Solver Implementation) specified by the parameters.
free_Helmholtz_2D/free_Helmholtz_3D/ free_sph_p/free_sph_np	Releases the memory used by the data structures needed for calling the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface[1].

[1]Poisson Solver routines call the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface for better performance.

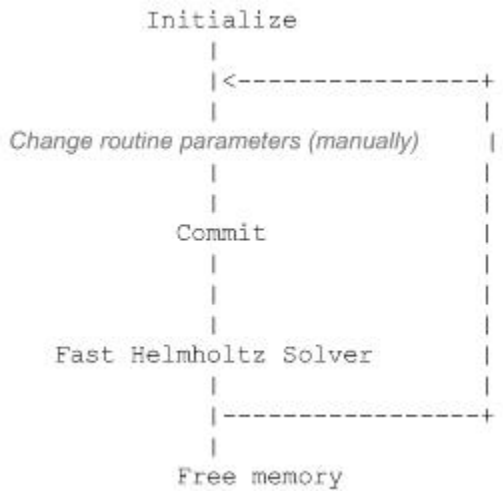
To find an approximate solution of Helmholtz problem only once, the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface routines are normally invoked in the order in which they are listed inTable "Poisson Solver Interface Routines".

NOTE
Though the order of invoking Poisson Solver routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in Figure "Typical Order of Invoking Poisson Solver Routines" indicates the typical order in which Poisson Solver routines can be invoked in a general case.

__border__top

Typical Order of Invoking Poisson Solver Routines



A general scheme of using Poisson Solver routines for double-precision computations in a 3D Cartesian case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below from the 3D to 2D case by changing the ending of the Poisson Solver routine names.

```
...
d_init_Helmholtz_3D(&ax, &bx, &ay, &by, &az, &bz, &nx, &ny, &nz, BCtype, ipar, dpar, &stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data
that is stored in f, save it to another location before the function call below */
d_commit_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar,
dpar, &stat);
d_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar, dpar,
&stat);
free_Helmholtz_3D (&xhandle, &yhandle, ipar, &stat);
/* here you may clean the memory used by f, dpar, ipar */
...
```

A general scheme of using Poisson Solver routines for double-precision computations in a spherical periodic case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below to a scheme for a non-periodic case by changing the ending of the Poisson Solver routine names from "p" to "np".

```
...
d_init_sph_p(&ap,&bp,&at,&bt,&np,&nt,&q,ipar,dpar,&stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data
that is stored in f, save it to another location before the function call below */
d_commit_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);
d_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);
free_sph_p(&handle_s,&handle_c,ipar,&stat);
/* here you may clean the memory used by f, dpar, ipar */
...
```

You can find examples of code that uses Poisson Solver routines to solve Helmholtz problem (in both Cartesian and spherical cases) in the `examples\pdepoissonf\source` folderin your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Fast Poisson Solver Interface Description

All numerical types in this section are either standard C types `float` and `double` or `MKL_INT` integer type. Fortran users can call the routines with `REAL` and `DOUBLE PRECISION` types of floating-point values and either `INTEGER` or `INTEGER*8` integer type depending on the programming interface (LP64 or ILP64). To better understand usage of the types, see examples in the `examples\pdepoissonf\source` folderin your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Routine Options

All Poisson Solver routines use parameters for passing various options to the routines. These parameters are arrays *ipar*, *dpar*, and *spar*. Values for these parameters should be specified very carefully (see Common Parameters). You can change these values during computations to meet your needs. For more details, see the descriptions of specific routines.

WARNING
To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.

User Data Arrays

Poisson Solver routines take arrays of user data as input. For example, the `d_Helmholtz_3D`routine takes user arrays to compute an approximate solution to the 3D Helmholtz problem. To minimize storage requirements and improve the overall run-time efficiency, Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver routines do not make copies of user input arrays.

NOTE
If you need a copy of your input data arrays, you must save them yourself.

For better performance, align your data arrays as recommended in the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* (search the document for coding techniques to improve performance).

Routines for the Cartesian Solver

The section describes Poisson Solver routines for the Cartesian case, their syntax, parameters, and return values. All flavors of the same routine are described together: single- and double-precision and 2D and 3D.

NOTE
Some of the routine parameters are used only in the 3D Fast Helmholtz Solver.

Poisson Solver routines call Intel® oneAPI Math Kernel Library (oneMKL) FFT routines (described inFFT Functions), which enhance performance of the Poisson Solver routines.

?_init_Helmholtz_2D/?_init_Helmholtz_3D

Initializes basic data structures of the Fast 2D/3D Helmholtz Solver.

Syntax

```
void d_init_Helmholtz_2D (const double * ax, const double * bx, const double * ay,
const double * by, const MKL_INT * nx, const MKL_INT * ny, const char * BCtype, const
double * q, MKL_INT * ipar, double * dpar, MKL_INT * stat);

void s_init_Helmholtz_2D (const float * ax, const float * bx, const float * ay, const
float * by, const MKL_INT * nx, const MKL_INT * ny, const char * BCtype, const float *
q, MKL_INT * ipar, float * spar, MKL_INT * stat);

void d_init_Helmholtz_3D (const double * ax, const double * bx, const double * ay,
const double * by, const double * az, const double * bz, const MKL_INT * nx, const
MKL_INT * ny, const MKL_INT * nz, const char * BCtype, const double * q, MKL_INT *ipar,
double * dpar, MKL_INT * stat);

void s_init_Helmholtz_3D (const float * ax, const float * bx, const float * ay, const
float * by, const float * az, const float * bz, const MKL_INT * nx, const MKL_INT * ny,
const MKL_INT * nz, const char * BCtype, const float * q, MKL_INT * ipar, float * spar,
MKL_INT * stat);
```

Include Files

- `mkl_poisson.f90`

Input Parameters

ax `double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`,

`float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`.

The coordinate of the leftmost boundary of the domain along the x-axis.

bx

`double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`,

`float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`.

The coordinate of the rightmost boundary of the domain along the x-axis.

ay

`double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`,

`float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`.

The coordinate of the leftmost boundary of the domain along the y-axis.

by

`double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`,

`float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`.

The coordinate of the rightmost boundary of the domain along the y-axis.

az

`double*` for `d_init_Helmholtz_3D`,

`float*` for `s_init_Helmholtz_3D`.

The coordinate of the leftmost boundary of the domain along the z-axis. This parameter is needed only for the `?_init_Helmholtz_3D` routine.

bz

`double*` for `d_init_Helmholtz_3D`,

`float*` for `s_init_Helmholtz_3D`.

The coordinate of the rightmost boundary of the domain along the z-axis. This parameter is needed only for the `?_init_Helmholtz_3D` routine.

nx

`MKL_INT*`. The number of mesh intervals along the x-axis.

ny

`MKL_INT*`. The number of mesh intervals along the y-axis.

nz

`MKL_INT*`. The number of mesh intervals along the z-axis. This parameter is needed only for the `?_init_Helmholtz_3D` routine.

BCtype

`char*`. Contains the type of boundary conditions on each boundary. Must contain four characters for `?_init_Helmholtz_2D` and six characters for `?_init_Helmholtz_3D`. Each of the characters can be 'N' (Neumann boundary condition), 'D' (Dirichlet boundary condition), or 'P' (periodic boundary conditions). Specify the types of boundary conditions for the boundaries in the following order: *bd_a*$_x$, *bd_b*$_x$, *bd_a*$_y$, *bd_b*$_y$, *bd_a*$_z$, and *bd_b*$_z$. Specify periodic boundary conditions on the respective boundaries in pairs (for example, 'PPDD' or 'NNPP' in the 2D case). The types of boundary conditions for the last two boundaries are needed only in the 3D case.

q

`double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`,

`float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D` .

	The constant Helmholtz coefficient. Note that to solve Poisson or Laplace problem, you should set the value of *q* to 0.

Output Parameters

ipar	MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to ipar).
dpar	double array of size 5*nx/2+7 in the 2D case or 5*(nx+ny)/2+9 in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).
spar	float array of size 5*nx/2+7 in the 2D case or 5*(nx+ny)/2+9 in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).
stat	MKL_INT*. Routine completion status, which is also written to *ipar[0]*. Continue to call other Poisson Solver routines only if the status is 0.

Description

The ?_init_Helmholtz_2D/?_init_Helmholtz_3D routines initialize basic data structures for Poisson Solver computations of the appropriate precision. All routines invoked after a call to a ?_init_Helmholtz_2D/?_init_Helmholtz_3D routine use values of the *ipar*, *dpar* and *spar* array parameters returned by the routine. Detailed description of the array parameters can be found in Common Parameters.

Caution
Data structures initialized and created by 2D flavors of the routine cannot be used by 3D flavors of any Poisson Solver routines, and vice versa.

You can skip calls to these routines in your code. However, see Caveat on Parameter Modifications for information on initializing the data structures.

Return Values

stat= 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this *stat* value.
stat= -99999	The routine failed to complete the task because of a fatal error.

_commit_Helmholtz_2D/?_commit_Helmholtz_3D

Checks consistency and correctness of input data and initializes certain data structures required to solve 2D/3D Helmholtz problem.

Syntax

```
void d_commit_Helmholtz_2D (double * f, const double * bd_ax, const double * bd_bx,
const double * bd_ay, const double * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT *
ipar, double * dpar, MKL_INT * stat );
```

```
void s_commit_Helmholtz_2D (float * f, const float * bd_ax, const float * bd_bx, const float * bd_ay, const float * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar, float * spar, MKL_INT * stat );
```

```
void d_commit_Helmholtz_3D (double * f, const double * bd_ax, const double * bd_bx, const double * bd_ay, const double * bd_by, const double * bd_az, const double * bd_bz, DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar, double * dpar, MKL_INT * stat );
```

```
void s_commit_Helmholtz_3D (float * f, const float * bd_ax, const float * bd_bx, const float * bd_ay, const float * bd_by, const float * bd_az, const float * bd_bz, DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar, float * spar, MKL_INT * stat );
```

Include Files

- mkl_poisson.f90

Input Parameters

f

`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`,

`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`.

Contains the right-hand side of the problem packed in a single vector:

- 2D problem: The size of the vector for the is (*nx*+1)*(*ny*+1). The value of the right-hand side in the mesh point (*i, j*) is stored in *f*[*i* +*j**(*nx*+1)] .
- 3D problem: The size of the vector for the is (*nx*+1)*(*ny*+1)*(*nz* +1). The value of the right-hand side in the mesh point (*i, j, k*) is stored in *f*[*i*+*j**(*nx*+1)+*k**(*nx*+1)*(*ny*+1)].

Note that to solve the Laplace problem, you should set all the elements of the array *f* to 0.

Note also that the array *f* may be altered by the routine. To preserve the *f* vector, save it to another memory location.

ipar

`MKL_INT` array of size 128. Contains integer data to be used by the Fast Helmholtz Solver (for details, refer to `ipar`).

dpar

`double` array of size depending on the dimension of the problem:

- 2D problem: 5**nx*/2+7
- 3D problem: 5*(*nx*+*ny*)/2+9

Contains double-precision data to be used by the Fast Helmholtz Solver (for details, refer to `dpar` and `spar`).

spar

`float` array of size depending on the dimension of the problem:

- 2D problem: 5**nx*/2+7
- 3D problem: 5*(*nx*+*ny*)/2+9

Contains single-precision data to be used by the Fast Helmholtz Solver (for details, refer to `dpar` and `spar`).

bd_ax

`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`,

`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`.

Contains values of the boundary condition on the leftmost boundary of the domain along the *x*-axis (for more information, refer to a detailed description of *bd_ax*).

bd_bx

`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`,

`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`.

Contains values of the boundary condition on the rightmost boundary of the domain along the *x*-axis (for more information, refer to a detailed description of *bd_bx*).

bd_ay

`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`,

`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`.

Contains values of the boundary condition on the leftmost boundary of the domain along the *y*-axis (for more information, refer to a detailed description of *bd_ay*).

bd_by

`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`,

`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`.

Contains values of the boundary condition on the rightmost boundary of the domain along the *y*-axis (for more information, refer to a detailed description of *bd_by*).

bd_az

`double*` for `d_commit_Helmholtz_3D`,

`float*` for `s_commit_Helmholtz_3D`.

Used only by `?_commit_Helmholtz_3D`. Contains values of the boundary condition on the leftmost boundary of the domain along the *z*-axis (for more information, refer to a detailed description of *bd_az*).

bd_bz

`double*` for `d_commit_Helmholtz_3D`,

`float*` for `s_commit_Helmholtz_3D`.

Used only by `?_commit_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along the *z*-axis (for more information, refer to a detailed description of *bd_bz*).

Output Parameters

f

Contains right-hand side of the problem, possibly altered on output.

ipar

Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in `ipar`.

dpar

Contains double-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in `dpar` and `spar`.

spar

Contains single-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in `dpar` and `spar`.

xhandle, yhandle

`DFTI_DESCRIPTOR_HANDLE*`. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer toFFT Functions). *yhandle* is used only by `?_commit_Helmholtz_3D`.

stat	MKL_INT*. Routine completion status, which is also written to *ipar[0]*. Continue to call other Poisson Solver routines only if the status is 0.

Description

The ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routines check the consistency and correctness of the parameters to be passed to the solver routines ?_Helmholtz_2D/?_Helmholtz_3D. They also initialize the *xhandle* and *yhandle* data structures, *ipar* array, and *dpar* or *spar* array, depending upon the routine precision. Refer to Common Parameters to find out which particular array elements the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routines initialize and to what values these elements are initialized.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of Poisson Solver routines, see Caveat on Parameter Modifications.
Unlike ?_init_Helmholtz_2D/?_init_Helmholtz_3D, you must call the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routines in your code. Values of *ax*, *bx*, *ay*, *by*, *az*, and *bz* are passed to the routines with the *spar/dpar* array, and values of *nx*, *ny*, *nz*, and *BCtype* are passed with the *ipar* array.

Return Values

stat= 1	The routine completed without errors but with warnings.
stat= 0	The routine successfully completed the task.
stat= -100	The routine stopped because an error in the input data was found, or the data in the *dpar*, *spar*, or *ipar* array was altered by mistake.
stat= -1000	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
stat= -10000	The routine stopped because the initialization failed to complete or the parameter *ipar*[0] was altered by mistake.
stat= -99999	The routine failed to complete the task because of a fatal error.

?_Helmholtz_2D/?_Helmholtz_3D

Computes the solution of the 2D/3D Helmholtz problem specified by the parameters.

Syntax

```
void d_Helmholtz_2D (double * f, const double * bd_ax, const double * bd_bx, const double * bd_ay, const double *bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar, const double * dpar, MKL_INT * stat );
```

```
void s_Helmholtz_2D (float * f, const float * bd_ax, const float * bd_bx, const float * bd_ay, const float * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar, const float * spar, MKL_INT * stat );
```

```
void d_Helmholtz_3D (double * f, const double * bd_ax, const double * bd_bx, const double * bd_ay, const double *bd_by, const double * bd_az, const double * bd_bz, DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar, const double * dpar, MKL_INT * stat );
```

```
void s_Helmholtz_3D (float * f, const float * bd_ax, const float * bd_bx, const float *
bd_ay, const float * bd_by, const float * bd_az, const float * bd_bz,
DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar,
const float * spar, MKL_INT * stat );
```

Include Files

- mkl_poisson.f90

Input Parameters

f

double* for d_Helmholtz_2D/d_Helmholtz_3D,

float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains the right-hand side of the problem packed in a single vector and modified by the appropriate ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine, at this point results in an incorrect solution.

- 2D problem: the size of the vector is (*nx*+1)*(*ny*+1). The value of the modified right-hand side in the mesh point (*i, j*) is stored in *f*[*i* +*j**(*nx*+1)].
- 3D problem: the size of the vector is (*nx*+1)*(*ny*+1)*(*nz*+1). The value of the modified right-hand side in the mesh point (*i, j, k*) is stored in *f*[*i*+*j**(*nx*+1)+*k**(*nx*+1)*(*ny*+1)].

xhandle, yhandle

DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer toFFT Functions). *yhandle* is used only by ?_Helmholtz_3D.

ipar

MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to ipar).

dpar

double array of size depending on the dimension of the problem:

- 2D problem: 5**nx*/2+7
- 3D problem: 5*(*nx*+*ny*)/2+9

Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).

spar

float array of size depending on the dimension of the problem:

- 2D problem: 5**nx*/2+7
- 3D problem: 5*(*nx*+*ny*)/2+9

Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).

bd_ax

double* for d_Helmholtz_2D/d_Helmholtz_3D,

float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains values of the boundary condition on the leftmost boundary of the domain along the *x*-axis (for more information, refer to a detailed description of *bd_ax*).

bd_bx
: `double*` for `d_Helmholtz_2D/d_Helmholtz_3D`,
`float*` for `s_Helmholtz_2D/s_Helmholtz_3D`.
Contains values of the boundary condition on the rightmost boundary of the domain along the *x*-axis (for more information, refer to a detailed description of *bd_bx*).

bd_ay
: `double*` for `d_Helmholtz_2D/d_Helmholtz_3D`,
`float*` for `s_Helmholtz_2D/s_Helmholtz_3D`.
Contains values of the boundary condition on the leftmost boundary of the domain along the *y*-axis for more information, refer to a detailed description of *bd_ay*).

bd_by
: `double*` for `d_Helmholtz_2D/d_Helmholtz_3D`,
`float*` for `s_Helmholtz_2D/s_Helmholtz_3D`.
Contains values of the boundary condition on the rightmost boundary of the domain along the *y*-axis (for more information, refer to a detailed description of *bd_by*).

bd_az
: `double*` for `d_Helmholtz_3D`,
`float*` for `s_Helmholtz_3D`.
Used only by `?_Helmholtz_3D`. Contains values of the boundary condition on the leftmost boundary of the domain along the *z*-axis (for more information, refer to a detailed description of *bd_az*).

bd_bz
: `double*` for `d_Helmholtz_3D`,
`float*` for `s_Helmholtz_3D`.
Used only by `?_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along the *z*-axis (for more information, refer to a detailed description of *bd_bz*).

NOTE
To avoid incorrect computation results, do not change arrays *bd_ax*, *bd_bx*, *bd_ay*, *bd_by*, *bd_az*, *bd_bz* between a call to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine and a subsequent call to the appropriate `?_Helmholtz_2D/?_Helmholtz_3D` routine.

Output Parameters

f
: On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.

xhandle, yhandle
: Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface. Although the addresses do not change, the structures are modified on output.

ipar
: Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in `ipar`.

stat	MKL_INT*. Routine completion status, which is also written to *ipar[0]*. Continue to call other Poisson Solver routines only if the status is 0.

Description

The ?_Helmholtz_2D/?_Helmholtz_3D routines compute the approximate solution of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to formulas given in Poisson Solver Implementation. The *f* parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of *ax*, *bx*, *ay*, *by*, *az*, and *bz* are passed to the routines with the *spar/dpar* array, and values of *nx*, *ny*, *nz*, and *BCtype* are passed with the *ipar* array.

Return Values

stat= 1	The routine completed without errors but with some warnings.
stat= 0	The routine successfully completed the task.
stat= -2	The routine stopped because division by zero occurred. It usually happens if the data in the *dpar* or *spar* array was altered by mistake.
stat= -3	The routine stopped because the sufficient memory was unavailable for the computations.
stat= -100	The routine stopped because an error in the input data was found or the data in the *dpar*, *spar*, or *ipar* array was altered by mistake.
stat= -1000	The routine stopped because of the Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
stat= -10000	The routine stopped because the initialization failed to complete or the parameter *ipar*[0] was altered by mistake.
stat= -99999	The routine failed to complete the task because of a fatal error.

free_Helmholtz_2D/free_Helmholtz_3D

Releases the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_Helmholtz_2D(DFTI_DESCRIPTOR_HANDLE* xhandle, MKL_INT* ipar, MKL_INT* stat);
void free_Helmholtz_3D(DFTI_DESCRIPTOR_HANDLE* xhandle, DFTI_DESCRIPTOR_HANDLE*
yhandle, MKL_INT* ipar, MKL_INT* stat);
```

Include Files

- mkl_poisson.f90

Input Parameters

xhandle, yhandle	DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer toFFT Functions). The structure *yhandle* is used only by free_Helmholtz_3D.
ipar	MKL_INT array of size 128. Contains integer data used by Fast Helmholtz Solver (for details, refer to ipar).

Output Parameters

xhandle, yhandle	Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface. Memory allocated for the structures is released on output.
ipar	Contains integer data used by Fast Helmholtz Solver. On output, the status of the routine call is written to *ipar[0]*.
stat	MKL_INT*. Routine completion status, which is also written to *ipar[0]*.

Description

The free_Helmholtz_2D-free_Helmholtz_3D routine releases the memory used by the *xhandle* and *yhandle*structures, which are needed for calling the Intel® oneAPI Math Kernel Library (oneMKL) FFT functions. To release memory allocated for other parameters, include memory release statements in your code.

Return Values

stat= 0	The routine successfully completed the task.
stat= -1000	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
stat= -99999	The routine failed to complete the task because of a fatal error.

Routines for the Spherical Solver

The section describes Poisson Solver routines for the spherical case, their syntax, parameters, and return values. All flavors of the same routine are described together: single- and double-precision and periodic (having names ending in "p") and non-periodic (having names ending in "np").

These Poisson Solver routines also call the Intel® oneAPI Math Kernel Library (oneMKL) FFT routines (described inFFT Functions), which enhance the performance of the Poisson Solver routines.

?_init_sph_p/?_init_sph_np

Initializes basic data structures of the periodic and non-periodic Fast Helmholtz Solver on a sphere.

Syntax

```
void d_init_sph_p (const double * ap, const double * at, const double * bp, const double * bt, const MKL_INT * np, const MKL_INT *nt, const double * q, MKL_INT * ipar, double * dpar, MKL_INT * stat );
```

```
void s_init_sph_p (const float * ap, const float * at, const float * bp, const float *
bt, const MKL_INT * np, const MKL_INT * nt, const float * q, MKL_INT * ipar, float *
spar, MKL_INT * stat );
```

```
void d_init_sph_np (const double * ap, const double * at, const double * bp, const
double * bt, const MKL_INT * np, const MKL_INT *nt, const double * q, MKL_INT * ipar,
double * dpar, MKL_INT * stat );
```

```
void s_init_sph_np (const float * ap, const float * at, const float * bp, const float *
bt, const MKL_INT * np, const MKL_INT * nt, const float * q, MKL_INT * ipar, float *
spar, MKL_INT * stat );
```

Include Files

- mkl_poisson.f90

Input Parameters

ap

`double*` for `d_init_sph_p/d_init_sph_np`,

`float*` for `s_init_sph_p/s_init_sph_np`.

The coordinate (angle) of the leftmost boundary of the domain along the φ-axis.

bp

`double*` for `d_init_sph_p/d_init_sph_np`,

`float*` for `s_init_sph_p/s_init_sph_np`.

The coordinate (angle) of the rightmost boundary of the domain along the φ-axis.

at

`double*` for `d_init_sph_p/d_init_sph_np`,

`float*` for `s_init_sph_p/s_init_sph_np`.

The coordinate (angle) of the leftmost boundary of the domain along the θ-axis.

bt

`double*` for `d_init_sph_p/d_init_sph_np`,

`float*` for `s_init_sph_p/s_init_sph_np`.

The coordinate (angle) of the rightmost boundary of the domain along the θ-axis.

np

`MKL_INT*`. The number of mesh intervals along the φ-axis. Must be even in the periodic case.

nt

`MKL_INT*`. The number of mesh intervals along the θ-axis.

q

`double*` for `d_init_sph_p/d_init_sph_np`,

`float*` for `s_init_sph_p/s_init_sph_np`.

The constant Helmholtz coefficient. To solve the Poisson problem, set the value of *q* to 0.

Output Parameters

ipar

`MKL_INT` array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to `ipar`).

dpar	`double` array of size 5*`np`/2+`nt`+10. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to `dpar` and `spar`).
spar	`float` array of size 5*`np`/2+`nt`+10. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to `dpar` and `spar`).
stat	`MKL_INT*`. Routine completion status, which is also written to *ipar[0]*. Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_init_sph_p/?_init_sph_np` routines initialize basic data structures for Poisson Solver computations. All routines invoked after a call to a `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine use values of the *ipar*, *dpar*, and *spar* array parameters returned by the routine. A detailed description of the array parameters can be found in Common Parameters.

> **Caution**
> Data structures initialized and created by periodic flavors of the routine cannot be used by non-periodic flavors of any Poisson Solver routines for Helmholtz Solver on a sphere, and vice versa.

You can skip calls to these routines in your code. However, see Caveat on Parameter Modifications for information on initializing the data structures.

Return Values

stat= 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this *stat* value.
stat= -99999	The routine failed to complete the task because of fatal error.

?_commit_sph_p/?_commit_sph_np

Checks consistency and correctness of input data and initializes certain data structures required to solve the periodic/non-periodic Helmholtz problem on a sphere.

Syntax

```
void d_commit_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s,
DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);
```

```
void s_commit_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE*
handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);
```

```
void d_commit_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double*
dpar, MKL_INT* stat);
```

```
void s_commit_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float*
spar, MKL_INT* stat);
```

Include Files

- mkl_poisson.f90

Input Parameters

f	double* for d_commit_sph_p/d_commit_sph_np, float* for s_commit_sph_p/s_commit_sph_np. Contains the right-hand side of the problem packed in a single vector. The size of the vector is (*np*+1)*(*nt*+1) and value of the right-hand side in the mesh point (*i, j*) is stored in *f*[*i*+*j**(*np*+1)] . Note that the array *f* may be altered by the routine. Save this vector to another memory location if you want to preserve it.
ipar	MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to ipar).
dpar	double array of size 5*np/2+nt+10. Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
spar	float array of size 5*np/2+nt+10. Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).

Output Parameters

f	Contains the right-hand side of the problem, possibly altered on output.
ipar	Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in ipar.
dpar	Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar.
spar	Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar.
handle_s, handle_c, handle	DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer toFFT Functions). *handle_s* and *handle_c* are used only in ?_commit_sph_p and *handle* is used only in ?_commit_sph_np.
stat	MKL_INT*. Routine completion status, which is also written to *ipar[0]*. Continue to call other Poisson Solver routines only if the status is 0.

Description

The ?_commit_sph_p/?_commit_sph_np routines check consistency and correctness of the parameters to be passed to the solver routines ?_sph_p/?_sph_np, respectively. They also initialize certain data structures. The routine ?_commit_sph_p initializes structures *handle_s* and *handle_c*, and ?_commit_sph_np initializes *handle*. The routines also initialize the *ipar* array and *dpar* or *spar* array,

depending upon the routine precision. Refer to Common Parameters to find out which particular array elements the ?_commit_sph_p/?_commit_sph_np routines initialize and to what values these elements are initialized.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of Poisson Solver routines, see Caveat on Parameter Modifications.
Unlike ?_init_sph_p/?_init_sph_np, you must call the ?_commit_sph_p/?_commit_sph_np routines. Values of *np* and *nt* are passed to each of the routines with the *ipar* array.

Return Values

stat= 1	The routine completed without errors but with warnings.
stat= 0	The routine successfully completed the task.
stat= -100	The routine stopped because an error in the input data was found or the data in the *dpar*, *spar*, or *ipar* array was altered by mistake.
stat= -1000	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
stat= -10000	The routine stopped because the initialization failed to complete or the parameter *ipar*[0] was altered by mistake.
stat= -99999	The routine failed to complete the task because of a fatal error.

?_sph_p/?_sph_np

Computes the solution of the spherical Helmholtz problem specified by the parameters.

Syntax

```
void d_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE*
handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);
```

```
void s_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE*
handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);
```

```
void d_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double* dpar,
MKL_INT* stat);
```

```
void s_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float* spar,
MKL_INT* stat);
```

Include Files

- mkl_poisson.f90

Input Parameters

f	double* for d_sph_p/d_sph_np, float* for s_sph_p/s_sph_np.

	Contains the right-hand side of the problem packed in a single vector and modified by the appropriate ?_commit_sph_p/?_commit_sph_np routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the ?_commit_sph_p/?_commit_sph_np routine, at this point results in an incorrect solution. The size of the vector is (*np*+1)*(*nt*+1) and the value of the modified right-hand side in the mesh point (*i, j*) is stored in *f*[*i*+*j**(*np*+1)] .
handle_s, handle_c, handle	DFTI_DESCRIPTOR_HANDLE*. Data structures used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer toFFT Functions). *handle_s* and *handle_c* are used only in ?_sph_p and *handle* is used only in ?_sph_np.
ipar	MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to ipar).
dpar	double array of size 5*np/2+nt+10. Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
spar	float array of size 5*np/2+nt+10. Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).

Output Parameters

f	On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.
handle_s, handle_c, handle	Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface.
ipar	Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in ipar.
dpar	Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar.
spar	Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar.
stat	MKL_INT*. Routine completion status, which is also written to *ipar[0]*. Continue to call other Poisson Solver routines only if the status is 0.

Description

The sph_p/sph_np routines compute the approximate solution on a sphere of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to the formulas given in Poisson Solver Implementation. The *f* parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of *np* and *nt* are passed to each of the routines with the *ipar* array.

Return Values

stat= 1	The routine completed without errors but with warnings.
stat= 0	The routine successfully completed the task.
stat= -2	The routine stopped because division by zero occurred. It usually happens if the data in the *dpar* or *spar* array was altered by mistake.
stat= -3	The routine stopped because the memory was insufficient to complete the computations.
stat= -100	The routine stopped because an error in the input data was found or the data in the *dpar*, *spar*, or *ipar* array was altered by mistake.
stat= -1000	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
stat= -10000	The routine stopped because the initialization failed to complete or the parameter *ipar*[0] was altered by mistake.
stat= -99999	The routine failed to complete the task because of a fatal error.

free_sph_p/free_sph_np

Releases the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_sph_p(DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c,
MKL_INT* ipar, MKL_INT* stat);
void free_sph_np(DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, MKL_INT* stat);
```

Include Files

- mkl_poisson.f90

Input Parameters

handle_s, handle_c, handle	DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer toFFT Functions). The structures *handle_s* and *handle_c* are used only in free_sph_p, and *handle* is used only in free_sph_np.
ipar	MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to ipar).

Output Parameters

handle_s, handle_c, handle	Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface. Memory allocated for the structures is released on output.

ipar	Contains integer data to be used by Fast Helmholtz Solver on a sphere. On output, the status of the routine call is written to *ipar[0]*.
stat	MKL_INT*. Routine completion status, which is also written to *ipar[0]*.

Description

The free_sph_p/free_sph_np routine releases the memory used by the *handle_s*, *handle_c* or *handle*structures, needed for calling the Intel® oneAPI Math Kernel Library (oneMKL) FFT functions. To release memory allocated for other parameters, include memory release statements in your code.

Return Values

stat= 0	The routine successfully completed the task.
stat= -1000	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
stat= -99999	The routine failed to complete the task because of a fatal error.

Common Parameters for the Poisson Solver

ipar

ipar	MKL_INT array of size 128, holds integer data needed for Fast Helmholtz Solver (both for Cartesian and spherical coordinate systems). Its elements are described in Table "Elements of the ipar Array":

NOTE
Initial values can be assigned to the array parameters by the appropriate ?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np and ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np routines.

Elements of the ipar Array

Index	Description
1	Contains status value of the last Poisson Solver routine called. In general, it should be 0 on exit from a routine to proceed with the Fast Helmholtz Solver. The element has no predefined values. This element can also be used to inform the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/ ?_commit_sph_p/?_commit_sph_np routines of how the Commit step of the computation should be carried out (see Figure "Typical Order of Invoking Poisson Solver Routines"). A non-zero value of *ipar(1)* with decimal representation $\overline{abc} = 100a + 10b + c$ =100*a*+10*b*+*c*, where each of *a*, *b*, and *c* is equal to 0 or 9, indicates that some parts of the Commit step should be omitted. • If *c*=9, the routine omits checking of parameters and initialization of the data structures. • If *b*=9,

Index	Description
	• In the Cartesian case, the routine omits the adjustment of the right-hand side vector *f* to the Neumann boundary condition (multiplication of boundary values by 0.5 as well as incorporation of the boundary function *g*) and/or the Dirichlet boundary condition (setting boundary values to 0 as well as incorporation of the boundary function *G*). • For the Helmholtz solver on a sphere, the routine omits computation of the spherical weights for the *dpar*/*spar* array. • If *a*=9, the routine omits the normalization of the right-hand side vector *f*. Depending on the solver, the normalization means: • 2D Cartesian case: multiplication by h_y^2, where h_y is the mesh size in the *y* direction (for details, see Poisson Solver Implementation). • 3D (Cartesian) case: multiplication by h_z^2, where h_z is the mesh size in the *z* direction. • Helmholtz solver on a sphere: multiplication by h_θ^2, where h_θ is the mesh size in the θ direction (for details, see Poisson Solver Implementation). Using *ipar(1)* you can adjust the routine to your needs and improve efficiency in solving multiple Helmholtz problems that differ only in the right-hand side. You must be cautious when using this method, because any misunderstanding of the commit process may cause incorrect results or program failure (see also Caveat on Parameter Modifications).
2	Contains error messaging options: • *ipar(2)*=-1 indicates that all error messages are printed to the MKL_Poisson_Library_log.txt file in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device (usually, screen). • *ipar(2)*=0 indicates that no error messages will be printed. • *ipar(2)*=1 is the default value. It indicates that all error messages are printed to the standard output device. In case of errors, the *stat* parameter contains a non-zero value on exit from a routine regardless of the *ipar(2)* setting.
3	Contains warning messaging options: • *ipar(3)*=-1 indicates that all warning messages are printed to the MKL_Poisson_Library_log.txt file in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • *ipar(3)*=0 indicates that no warning messages will be printed. • *ipar(3)*=1 is the default value. It indicates that all warning messages are printed to the standard output device. In case of warnings, the *stat* parameter contains a non-zero value on exit from a routine regardless of the *ipar(3)* setting.
4 through 6	Internal parameters.
Parameters 7 through 12 are used only in the Cartesian case.	
7	Takes this value: • 2, if *BCtype[0]*='P' • 1, if *BCtype[0]*='N' • 0, if *BCtype[0]*='D' • -1, otherwise
8	Takes this value: • 2, if *BCtype[1]*='P'

Index	Description
	• 1, if *BCtype[1]*='N' • 0, if *BCtype[1]*='D' • -1, otherwise
9	Takes this value: • 2, if *BCtype[2]*='P' • 1, if *BCtype[2]*='N' • 0, if *BCtype[2]*='D' • -1, otherwise
10	Takes this value: • 2, if *BCtype[3]*='P' • 1, if *BCtype[3]*='N' • 0, if *BCtype[3]*='D' • -1, otherwise
11	Takes this value: • 2, if *BCtype[4]*='P' • 1, if *BCtype[4]*='N' • 0, if *BCtype[4]*='D' • -1, otherwise
12	Takes this value: • 2, if *BCtype[5]*='P' • 1, if *BCtype[5]*='N' • 0, if *BCtype[5]*='D' • -1, otherwise
13	Takes the value of • *nx*, that is, the number of intervals along the *x*-axis, in the Cartesian case. • *np*, that is, the number of intervals along the φ-axis, in the spherical case.
14	Takes the value of • *ny*, that is, the number of intervals along the *y*-axis, in the Cartesian case • *nt*, that is, the number of intervals along the θ-axis, in the spherical case.
15	Takes the value of *nz*, the number of intervals along the *z*-axis. This parameter is used only in the 3D case (Cartesian).
16 through 23	Internal parameters which define the internal partitioning of the *dpar*/*spar* array.

The values of *ipar(22)* - *ipar(120)* are assigned regardless of the dimension of the problem for the Cartesian solver or of whether the solver on a sphere is periodic.

24	Contains message style options: • *ipar(22)*=0 indicates that Poisson Solver routines prints all error and warning messages in Fortran-style notations.
25	Contains the number of OpenMP threads to be used for computations in a multithreaded environment. The default value is 1 in the serial mode, and the result returned by the mkl_get_max_threads function otherwise.

Index	Description
26 through 29	Internal parameters which define the internal partitioning of the *dpar*/*spar* array.
26	Takes the value of *ipar(19)*+1, which specifies the internal partitioning of the *dpar/spar* array in the periodic Cartesian case.
27	Takes the value of *ipar(24)*+3**ipar(13)*/4, which specifies the internal partitioning of the *dpar/spar* array in the periodic Cartesian case.
28	Takes the value of *ipar(21)*+1, which specifies the internal partitioning of the *dpar/spar* array in the periodic 3D Cartesian case.
29	Takes the value of *ipar(26)*+3**ipar(14)*/4, which specifies the internal partitioning of the *dpar/spar* array in the periodic 3D Cartesian case.
30 through 40	Unused.
41 through 60	Contain the first twenty elements of the *ipar* array of the first Trigonometric Transform that the solver uses. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
61 through 80	Contain the first twenty elements of the *ipar* array of the second Trigonometric Transform that the 3D Cartesian and periodic spherical solvers use. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
81 through 100	Contain the first twenty elements of the *ipar* array of the third Trigonometric Transform that the solver uses in case of periodic boundary conditions along the *x*-axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
101 through 120	Contain the first twenty elements of the *ipar* array of the fourth Trigonometric Transform used by periodic spherical solvers and 3D Cartesian solvers with periodic boundary conditions along the *y*-axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
121 through 129	Internal parameters used by nonuniform 3D solvers.

NOTE
While you can declare the *ipar* array as `MKL_INT ipar(121)`, for future compatibility you should declare *ipar* as `MKL_INT ipar(129)`.

dpar and spar

Arrays *dpar* and *spar* are the same except in the data precision:

dpar Holds data needed for double-precision Fast Helmholtz Solver computations.

- For the Cartesian solver, `double` array of size 5**nx*/2+7 in the 2D case or 5*(*nx*+*ny*)/2+9 in the 3D case; initialized in the `d_init_Helmholtz_2D/d_init_Helmholtz_3D` and `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D` routines.

- For the spherical solver, `double` array of size 5*np/2+nt+10; initialized in the `d_init_sph_p/d_init_sph_np` and `d_commit_sph_p/d_commit_sph_np` routines.

spar

Holds data needed for single-precision Fast Helmholtz Solver computations.

- For the Cartesian solver, `float` array of size 5*nx/2+7 in the 2D case or 5*(nx+ny)/2+9 in the 3D case; initialized in the `s_init_Helmholtz_2D/s_init_Helmholtz_3D` and `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D` routines.
- For the spherical solver, `float` array of size 5*np/2+nt+10; initialized in the `s_init_sph_p/s_init_sph_np` and `s_commit_sph_p/s_commit_sph_np` routines.

Because *dpar* and *spar* have similar elements in each position, the elements are described together in Table "Elements of the `dpar` and `spar` Arrays":

Elements of the dpar and spar Arrays

Index	Description
1	In the Cartesian case, contains the length of the interval along the *x*-axis right after a call to the `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine or the mesh size h_x in the *x* direction (for details, see Poisson Solver Implementation) after a call to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine. In the spherical case, contains the length of the interval along the φ-axis right after a call to the `?_init_sph_p/?_init_sph_np` routine or the mesh size h_φ in the φ direction (for details, see Poisson Solver Implementation) after a call to the `?_commit_sph_p/?_commit_sph_np` routine.
2	In the Cartesian case, contains the length of the interval along the *y*-axis right after a call to the `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine or the mesh size h_y in the *y* direction (for details, see Poisson Solver Implementation) after a call to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine. In the spherical case, contains the length of the interval along the θ-axis right after a call to the `?_init_sph_p/?_init_sph_np` routine or the mesh size h_θ in the θ direction (for details, see Poisson Solver Implementation) after a call to the `?_commit_sph_p/?_commit_sph_np` routine.
3	In the Cartesian case, contains the length of the interval along the *z*-axis right after a call to the `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine or the mesh size h_z in the *z* direction (for details, see Poisson Solver Implementation) after a call to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine. In the Cartesian solver, this parameter is used only in the 3D case. In the spherical solver, contains the coordinate of the leftmost boundary along the θ-axis after a call to the `?_init_sph_p/?_init_sph_np` routine.
4	Contains the value of the coefficient q after a call to the `?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np` routine.
5	Contains the tolerance parameter after a call to the `?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np` routine.

Index	Description
	• In the Cartesian case, this value is used only for the pure Neumann boundary conditions (*BCtype*="NNNN" in the 2D case; *BCtype*="NNNNNN" in the 3D case). This is a special case, because the right-hand side of the problem cannot be arbitrary if the coefficient *q* is zero. The Poisson Solver verifies that the classical solution exists (up to rounding errors) using this tolerance. In any case, the Poisson Solver computes the normal solution, that is, the solution that has the minimal Euclidean norm of residual. Nevertheless, the ?_Helmholtz_2D/?_Helmholtz_3D routine informs you that the solution may not exist in a classical sense (up to rounding errors). • In the spherical case, the value is used for the special case of a periodic problem on the entire sphere. This special case is similar to the Cartesian case with pure Neumann boundary conditions. Here the Poisson Solver computes the normal solution as well. The parameter is also used to detect whether the problem is periodic up to rounding errors. The default value for this parameter is 1.0E-10 in case of double-precision computations or 1.0E-4 in case of single-precision computations. You can increase the value of the tolerance, for instance, to avoid the warnings that may appear.
ipar(16) through *ipar(17)*	In the Cartesian case, contain the spectrum of the one-dimensional (1D) problem along the *x*-axis after a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. In the spherical case, contains the spectrum of the 1D problem along the φ-axis after a call to the ?_commit_sph_p/?_commit_sph_np routine.
ipar(18) through *ipar(19)*	In the Cartesian case, contain the spectrum of the 1D problem along the *y*-axis after a call to the ?_commit_Helmholtz_3D routine. These elements are used only in the 3D case. In the spherical case, contains the spherical weights after a call to the ?_commit_sph_p/?_commit_sph_np routine.
ipar(20) through *ipar(21)*	Take the values of the (staggered) sine/cosine in the mesh points: • along the *x*-axis after a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine for a Cartesian solver • along the φ-axis after a call to the ?_commit_sph_p/?_commit_sph_np routine for a spherical solver.
ipar(22) through *ipar(23)*	Take the values of the (staggered) sine/cosine in the mesh points: • along the *y*-axis after a call to the ?_commit_Helmholtz_3D routine for a Cartesian 3D solver • along the φ-axis after a call to the ?_commit_sph_p routine for a spherical periodic solver. These elements are not used in the 2D Cartesian case and in the non-periodic spherical case.
ipar(26) through *ipar(27)*	Take the values of the (staggered) sine/cosine in the mesh points along the *x*-axis after a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. These elements are used only in the periodic Cartesian case.
ipar(28) through *ipar(29)-1*	Take the values of the (staggered) sine/cosine in the mesh points along the *x*-axis after a call to the ?_commit_Helmholtz_3D routine. These elements are used only in the periodic 3D Cartesian case.

NOTE
You may define the array size depending upon the type of the problem to solve.

Caveat on Parameter Modifications

Flexibility of the Poisson Solver interface enables you to skip calls to the ?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np routine and to initialize the basic data structures explicitly in your code. You may also need to modify contents of the *ipar*, *dpar*, and *spar* arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or incorrect results. You can perform a basic check for correctness and consistency of parameters by calling the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine; however, this does not ensure the correct solution but only reduces the chance of errors or wrong results.

NOTE
To supply correct and consistent parameters to Poisson Solver routines, you should have considerable experience in using the Poisson Solver interface and good understanding of the solution process, as well as elements contained in the *ipar*, *spar*, and *dpar* arrays and dependencies between values of these elements.

In rare occurrences when you fail in tuning parameters for the Fast Helmholtz Solver, refer for technical support at http://www.intel.com/software/products/support/ .

WARNING
The only way that ensures a proper solution of a Helmholtz problem is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of *ipar*, *dpar*, and *spar* arrays unless it is necessary.

Parameters That Define Boundary Conditions

Poisson Solver routines for the Cartesian solver use the following common parameters to define the boundary conditions.

Parameters to Define Boundary Conditions for the Cartesian Solver

Parameter	Description

bd_ax	`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D` and `d_Helmholtz_2D/d_Helmholtz_3D`,
	`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D` and `s_Helmholtz_2D/s_Helmholtz_3D`.
	Contains values of the boundary condition on the leftmost boundary of the domain along the *x*-axis.
	• 2D problem: the size of the array is *ny*+1. Its contents depend on the boundary conditions as follows:
	  • Dirichlet boundary condition (value of *BCtype*[0] is 'D'): values of the function $G(ax, y_j)$, j=0, ..., *ny*.
	  • Neumann boundary condition (value of *BCtype*[0] is 'N'): values of the function $g(ax, y_j)$, j=0, ..., *ny*.
	  The value corresponding to the index j is placed in *bd_ax*[j].
	• 3D problem: the size of the array is (*ny*+1)*(*nz*+1). Its contents depend on the boundary conditions as follows:

Parameter	Description
	• Dirichlet boundary condition (value of *BCtype*[0] is 'D'): values of the function $G(ax, y_j, z_k)$, j=0, ..., *ny*, k=0, ..., *nz*. • Neumann boundary condition (value of *BCtype*[0] is 'N'): the values of the function $g(ax, y_j, z_k)$, j=0, ..., *ny*, k=0, ..., *nz*. The values are packed in the array so that the value corresponding to indices *(j, k)* is placed in *bd_ax*[$j+k*(ny+1)$].
	For periodic boundary conditions (the value of *BCtype*[0] is 'P'), this parameter is not used, so it can accept a dummy pointer.

bd_bx	`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D` and `d_Helmholtz_2D/d_Helmholtz_3D`,
	`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D` and `s_Helmholtz_2D/s_Helmholtz_3D`.
	Contains values of the boundary condition on the rightmost boundary of the domain along the *x*-axis.
	• 2D problem: the size of the array is *ny*+1. Its contents depend on the boundary conditions as follows:
	  • Dirichlet boundary condition (value of *BCtype*[1] is 'D'): values of the function $G(bx, y_j)$, j=0, ..., *ny*.
	  • Neumann boundary condition (value of *BCtype*[1] is 'N'): values of the function $g(bx, y_j)$, j=0, ..., *ny*.
	  The value corresponding to the index j is placed in *bd_bx*[j].
	• 3D problem: the size of the array is (*ny*+1)*(*nz*+1). Its contents depend on the boundary conditions as follows:
	  • Dirichlet boundary condition (value of *BCtype*[1] is 'D'): values of the function $G(bx, y_j, z_k)$, j=0, ..., *ny*, k=0, ..., *nz*.
	  • Neumann boundary condition (value of *BCtype*[1] is 'N'): values of the function $g(bx, y_j, z_k)$, j=0, ..., *ny*, k=0, ..., *nz*.
	  The values are packed in the array so that the value corresponding to indices *(j, k)* is placed in *bd_bx*[$j+k*(ny+1)$].
	For periodic boundary conditions (the value of *BCtype*[1] is 'P'), this parameter is not used, so it can accept a dummy pointer.

bd_ay	`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D` and `d_Helmholtz_2D/d_Helmholtz_3D`,
	`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D` and `s_Helmholtz_2D/s_Helmholtz_3D`.
	Contains values of the boundary condition on the leftmost boundary of the domain along the *y*-axis.
	• 2D problem: the size of the array is *nx*+1. Its contents depend on the boundary conditions as follows:
	  • Dirichlet boundary condition (value of *BCtype*[2] is 'D'): values of the function $G(x_i, ay)$, i=0, ..., *nx*.
	  • Neumann boundary condition (value of *BCtype*[2] is 'N'): values of the function $g(x_i, ay)$, i=0, ..., *nx*.
	  The value corresponding to the index i is placed in *bd_ay*[i].
	• 3D problem: the size of the array is (*nx*+1)*(*nz*+1). Its contents depend on the boundary conditions as follows:

Parameter	Description
	• Dirichlet boundary condition (value of *BCtype*[2] is 'D'): values of the function $G(x_i, ay, z_k)$, i=0, ..., *nx*, k=0, ..., *nz*. • Neumann boundary condition (value of *BCtype*[2] is 'N'): values of the function $g(x_i, ay, z_k)$, i=0, ..., *nx*, k=0, ..., *nz*. The values are packed in the array so that the value corresponding to indices *(i, k)* is placed in *bd_ay*[*i*+*k**(*nx*+1)]. For periodic boundary conditions (the value of *BCtype*[2] is 'P'), this parameter is not used, so it can accept a dummy pointer.

bd_by	`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D` and `d_Helmholtz_2D/d_Helmholtz_3D`,
	`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D` and `s_Helmholtz_2D/s_Helmholtz_3D`.
	Contains values of the boundary condition on the rightmost boundary of the domain along the *y*-axis.
	• 2D problem: the size of the array is *nx*+1. Its contents depend on the boundary conditions as follows:
	  • Dirichlet boundary condition (value of *BCtype*[3] is 'D'): values of the function $G(x_i, by)$, i=0, ..., *nx*.
	  • Neumann boundary condition (value of *BCtype*[3] is 'N'): values of the function $g(x_i, by)$, i=0, ..., *nx*.
	  The value corresponding to the index *i* is placed in *bd_by*[*i*].
	• 3D problem: the size of the array is (*nx*+1)*(*nz*+1). Its contents depend on the boundary conditions as follows:
	  • Dirichlet boundary condition (value of *BCtype*[3] is 'D'): values of the function $G(x_i, by, z_k)$, i=0, ..., *nx*, k=0, ..., *nz*.
	  • Neumann boundary condition (value of *BCtype*[3] is 'N'): values of the function $g(x_i, by, z_k)$, i=0, ..., *nx*, k=0, ..., *nz*.
	  The values are packed in the array so that the value corresponding to indices *(i, k)* is placed in *bd_by*[*i*+*k**(*nx*+1)].
	For periodic boundary conditions (the value of *BCtype*[3] is 'P'), this parameter is not used, so it can accept a dummy pointer.

bd_az	`double*` for `d_commit_Helmholtz_3D` and `d_Helmholtz_3D`, `float*` for `s_commit_Helmholtz_3D` and `s_Helmholtz_3D`. Used only by `?_commit_Helmholtz_3D` and `?_Helmholtz_3D`. Contains values of the boundary condition on the leftmost boundary of the domain along the *z*-axis. The size of the array is (*nx*+1)*(*ny*+1). Its contents depend on the boundary conditions as follows: • Dirichlet boundary condition (value of *BCtype*[4] is 'D'): values of the function $G(x_i, y_j, az)$, i=0, ..., *nx*, j=0, ..., *ny*. • Neumann boundary condition (value of *BCtype*[4] is 'N'), values of the function $g(x_i, y_j, az)$, i=0, ..., *nx*, j=0, ..., *ny*. The values are packed in the array so that the value corresponding to indices *(i, j)* is placed in *bd_az*[*i*+*j**(*nx*+1)]. For periodic boundary conditions (the value of *BCtype*[4] is 'P'), this parameter is not used, so it can accept a dummy pointer.

Parameter	Description
bd_bz	`double*` for `d_commit_Helmholtz_3D` and `d_Helmholtz_3D`, `float*` for `s_commit_Helmholtz_3D` and `s_Helmholtz_3D`. Used only by `?_commit_Helmholtz_3D` and `?_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along the *z*-axis. The size of the array is (*nx*+1)*(*ny*+1). Its contents depend on the boundary conditions as follows: • Dirichlet boundary condition (value of *BCtype*[5] is 'D'): values of the function $G(x_i, y_j, bz)$, i=0, ..., *nx*, j=0, ..., *ny*. • Neumann boundary condition (value of *BCtype*[5] is 'N'): values of the function $g(x_i, y_j, bz)$, i=0, ..., *nx*, j=0, ..., *ny*. The values are packed in the array so that the value corresponding to indices *(i, j)* is placed in *bd_bz*[*i*+*j**(*nx*+1)]. For periodic boundary conditions (the value of *BCtype*[5] is 'P'), this parameter is not used, so it can accept a dummy pointer.

See Also

`?_commit_Helmholtz_2D/?_commit_Helmholtz_3D`

`?_Helmholtz_2D/?_Helmholtz_3D`

Poisson Solver Implementation Details

Several aspects of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface are platform-specific and language-specific. To promote portability of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface across platforms and ease of use across different languages, Intel® oneAPI Math Kernel Library (oneMKL) provides you with the Poisson Solver language-specific header file to include in your code:

- `mkl_poisson.f90`, to be used together with `mkl_dfti.f90`.

NOTE

- Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface supports Fortran versions starting with Fortran 90.
- Use of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver software without including the above language-specific header files is not supported.

Header File

The header file defines the following function prototypes for the Cartesian solver:

```
SUBROUTINE D_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR, DPAR, STAT)
   USE MKL_DFTI

   INTEGER NX, NY, STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION AX, BX, AY, BY, Q
   DOUBLE PRECISION DPAR(*)
   CHARACTER(4) BCTYPE
END SUBROUTINE


SUBROUTINE D_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, DPAR, STAT)
   USE MKL_DFTI
```

```
   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,*)
   DOUBLE PRECISION DPAR(*)
   DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE


SUBROUTINE D_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, DPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,*)
   DOUBLE PRECISION DPAR(*)
   DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE


SUBROUTINE S_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR, SPAR, STAT)
   USE MKL_DFTI

   INTEGER NX, NY, STAT
   INTEGER IPAR(*)
   REAL AX, BX, AY, BY, Q
   REAL SPAR(*)
   CHARACTER(4) BCTYPE
END SUBROUTINE


SUBROUTINE S_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, SPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL F(IPAR(11)+1,*)
   REAL SPAR(*)
   REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE


SUBROUTINE S_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, SPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL F(IPAR(11)+1,*)
   REAL SPAR(*)
   REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE


SUBROUTINE FREE_HELMHOLTZ_2D (XHANDLE, IPAR, STAT)
```

```
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE



SUBROUTINE D_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE, Q, IPAR, DPAR, STAT)
   USE MKL_DFTI

   INTEGER NX, NY, NZ, STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION AX, BX, AY, BY, AZ, BZ, Q
   DOUBLE PRECISION DPAR(*)
   CHARACTER(6) BCTYPE
END SUBROUTINE


SUBROUTINE D_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE,
IPAR, DPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,IPAR(12)+1,*)
   DOUBLE PRECISION DPAR(*)
   DOUBLE PRECISION BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
   DOUBLE PRECISION BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE


SUBROUTINE D_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR,
DPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,IPAR(12)+1,*)
   DOUBLE PRECISION DPAR(*)
   DOUBLE PRECISION BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
   DOUBLE PRECISION BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE


SUBROUTINE S_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE, Q, IPAR, SPAR, STAT)
   USE MKL_DFTI

   INTEGER NX, NY, NZ, STAT
   INTEGER IPAR(*)
   REAL AX, BX, AY, BY, AZ, BZ, Q
   REAL SPAR(*)
   CHARACTER(6) BCTYPE
END SUBROUTINE
```

```
SUBROUTINE S_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE,
IPAR, SPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL F(IPAR(11)+1,IPAR(12)+1,*)
   REAL SPAR(*)
   REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
   REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE


SUBROUTINE S_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR,
SPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL F(IPAR(11)+1,IPAR(12)+1,*)
   REAL SPAR(*)
   REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
   REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE


SUBROUTINE FREE_HELMHOLTZ_3D (XHANDLE, YHANDLE, IPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE
```

The header file defines the following function prototypes for the spherical solver:

```
SUBROUTINE D_INIT_SPH_P(AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER NP, NT, STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION AP,BP,AT,BT,Q
   DOUBLE PRECISION DPAR(*)
END SUBROUTINE


SUBROUTINE D_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION DPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE
```

```
SUBROUTINE D_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION DPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE


SUBROUTINE S_INIT_SPH_P(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER NP, NT, STAT
   INTEGER IPAR(*)
   REAL AP,BP,AT,BT,Q
   REAL SPAR(*)
END SUBROUTINE


SUBROUTINE S_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL SPAR(*)
   REAL F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE


SUBROUTINE S_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL SPAR(*)
   REAL F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE


SUBROUTINE FREE_SPH_P(HANDLE_S,HANDLE_C,IPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_S, HANDLE_C
END SUBROUTINE



SUBROUTINE D_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
   USE MKL_DFTI
```

```
   INTEGER NP, NT, STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION AP,BP,AT,BT,Q
   DOUBLE PRECISION DPAR(*)
END SUBROUTINE


SUBROUTINE D_COMMIT_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION DPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE


SUBROUTINE D_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION DPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE


SUBROUTINE S_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER NP, NT, STAT
   INTEGER IPAR(*)
   REAL AP,BP,AT,BT,Q
   REAL SPAR(*)
END SUBROUTINE


SUBROUTINE S_COMMIT_SPH_NP(F,HANDLE,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL SPAR(*)
   REAL F(IPAR(11)+1,*)
     TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE


SUBROUTINE S_SPH_NP(F,HANDLE,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL SPAR(*)
   REAL F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
```

```
END SUBROUTINE


SUBROUTINE FREE_SPH_NP(HANDLE,IPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE
```

Fortran specifics of the Poisson Solver routines usage are similar for all Intel® oneAPI Math Kernel Library (oneMKL) PDE support tools and described inCalling PDE Support Routines from Fortran.

Calling PDE Support Routines from Fortran

The calling interface for all the Intel® oneAPI Math Kernel Library (oneMKL) TT and Poisson Solver routines is designed to be easily used in C. However, you can invoke each TT or Poisson Solver routine directly from Fortran 90 or higher if you are familiar with the inter-language calling conventions of your platform.

The TT or Poisson Solver interface cannot be invoked from FORTRAN 77 due to restrictions imposed by the use of the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface.

The inter-language calling conventions include, but are not limited to, the argument passing mechanisms for the language, the data type mappings from C to Fortran, and how C external names are decorated on the platform.

To promote portability and relieve you of dealing with the calling conventions specifics, the Fortran header file `mkl_trig_transforms.f90` for TT routines and `mkl_poisson.f90` for Poisson Solver routines, used together with `mkl_dfti.f90`, declare a set of macros and introduce type definitions intended to hide the inter-language calling conventions and provide an interface to the routines that looks natural in Fortran.

For example, consider a hypothetical library routine, `foo`, which takes a double-precision vector of length *n*. C users access such a function as follows:

```
MKL_INT n;
  double *x;
  …
  foo(x, &n);
```

As noted above, to invoke `foo`, Fortran users would need to know what Fortran data types correspond to C types `MKL_INT` and `double` (or `float` for single-precision), what argument-passing mechanism the C compiler uses and what, if any, name decoration is performed by the C compiler when generating the external symbol `foo`. However, with the Fortran header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` included, the invocation of `foo` within a Fortran program will look as follows for the LP64 interface (for the ILP64 interface, `INTEGER*8` type will be used instead of `INTEGER*4`):

- For TT interface,

```
use mkl_dfti
use mkl_trig_transforms
INTEGER*4 n
DOUBLE PRECISION, ALLOCATABLE :: x
…
CALL FOO(x,n)
```

- For Poisson Solver interface,

```
use mkl_dfti
use mkl_poisson
INTEGER*4 n
DOUBLE PRECISION, ALLOCATABLE :: x
```

```
…
CALL FOO(x,n)
```

Note that in the above example, the header files `mkl_trig_transforms.f90 / mkl_poisson.f90` and `mkl_dfti.f90` provide a definition for the subroutine `FOO`. To ease the use of Poisson Solver or TT routines in Fortran, the general approach of providing Fortran definitions of names is used throughout the libraries. Specifically, if a name from a Poisson Solver or TT interface is documented as having the C-specific name `foo`, then the Fortran header files provide an appropriate Fortran language type definition `FOO`.

One of the key differences between Fortran and C is the language argument-passing mechanism: C programs use pass-by-value semantics and Fortran programs use pass-by-reference semantics. The Fortran headers ensure proper treatment of this difference. In particular, in the above example, the header files `mkl_trig_transforms.f90 / mkl_poisson.f90` and `mkl_dfti.f90` hide the difference by defining a macro `FOO` that takes the address of the appropriate arguments.

Nonlinear Optimization Problem Solvers

Intel® oneAPI Math Kernel Library (oneMKL) provides tools for solving nonlinear least squares problems using the Trust-Region (TR) algorithms. The general nonlinear solver workflow and naming conventions are described here:

- Nonlinear Solver Organization and Implementation
- Nonlinear Solver Routine Naming Conventions

The solver routines are grouped according to their purpose as follows:

- Nonlinear Least Squares Problem without Constraints
- Nonlinear Least Squares Problem with Linear (Boundary) Constraints
- Jacobian Matrix Calculation Routines

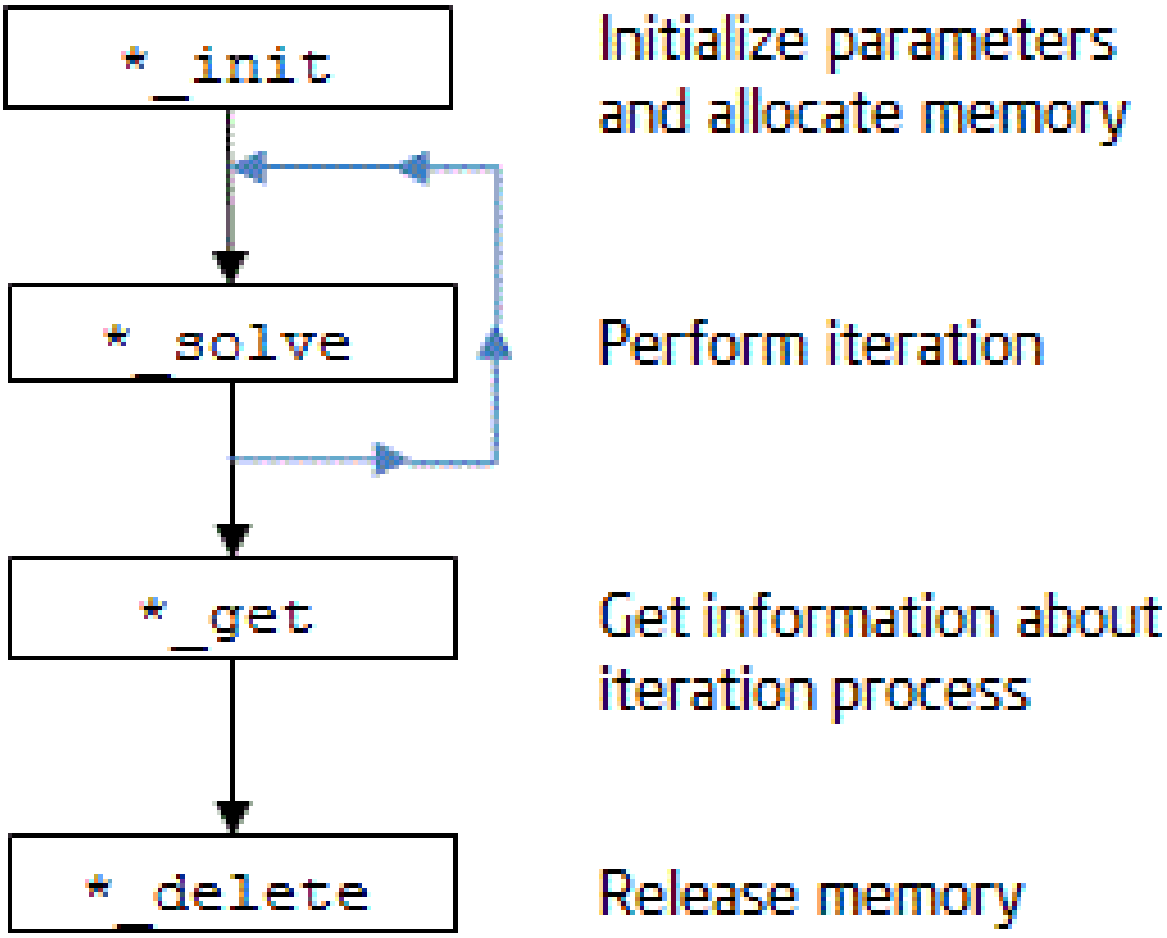
For more information on the key concepts required to understand the use of the Intel® oneAPI Math Kernel Library (oneMKL) nonlinear least squares problem solver routines, see [Conn00].

Nonlinear Solver Organization and Implementation

The Intel® oneAPI Math Kernel Library (oneMKL) solver routines for nonlinear least squares problems use reverse communication interfaces (RCI). That means you need to provide the solver with information required for the iteration process, for example, the corresponding Jacobian matrix, or values of the objective function. RCI removes the dependency of the solver on specific implementation of the operations. However, it does require that you organize a computational loop.

__border__top

Typical order for invoking RCI solver routines



The nonlinear least squares problem solver routines, or Trust-Region (TR) solvers, are implemented with threading support. You can manage the threads using Threading Control Functions. The TR solvers use BLAS and LAPACK routines, and offer the same parallelism as those domains. The `?jacobi` and `?jacobix` routines of Jacobi matrix calculations are parallel. These routines (`?jacobi` and `?jacobix`) make calls to the user-supplied functions with different `x` parameters for multiple threads.

Memory Allocation and Handles

To make the TR solver routines easy to use, you are not required to allocate temporary working storage. The solver allocates all temporary memory internally. To allow multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using a data object called a handle. Each TR solver routine creates, uses, or deletes a handle. The handle datatype definition can be found in `mkl_rci.fi` (or `mkl_rci.f90`).

Use one of the mentioned headers (either include it or as a module) and declare the handle as:

```
type(HANDLE_TR) :: handle
```

For a program using compilers that support eight byte integers, declare a handle as:

```
INCLUDE "mkl_rci.fi"
INTEGER*8 handle
```

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Nonlinear Solver Routine Naming Conventions

The TR routine names have the following structure:

```
      <character><name>_<action>( )
```

where

- <*character*> indicates the data type:

`s`	real, single precision
`d`	real, double precision

- <*name*> indicates the task type:

`trnlsp`	nonlinear least squares problem without constraints
`trnlspbc`	nonlinear least squares problem with boundary constraints
`jacobi`	computation of the Jacobian matrix using central differences

- <*action*> indicates an action on the task:

`init`	initializes the solver
`check`	checks correctness of the input parameters
`solve`	solves the problem
`get`	retrieves the number of iterations, the stop criterion, the initial residual, and the final residual
`delete`	releases the allocated data

Nonlinear Least Squares Problem without Constraints

The nonlinear least squares problem without constraints can be described as follows:

$$\min_{x \in R^n} \left\| F(x) \right\|_2^2 = \min_{x \in R^n} \left\| y - f(x) \right\|_2^2, y \in R^m, x \in R^n, f : R^n \to R^m, m \geq n,$$

where

$F(x) : R^n \to R^m$ is a twice differentiable function in R^n.

Solving a nonlinear least squares problem means searching for the best approximation to the vector y with the model function $f_i(x)$ and nonlinear variables x. The best approximation means that the sum of squares of residuals $y_i - f_i(x)$ is the minimum.

See usage examples in the `examples\f\nonlinear_solvers` folderof your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see `ex_nlsqp_f.f`.

RCI TR Routines

Routine Name	Operation
?trnlsp_init	Initializes the solver.
?trnlsp_check	Checks correctness of the input parameters.
?trnlsp_solve	Solves a nonlinear least squares problem using the Trust-Region algorithm.
?trnlsp_get	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
?trnlsp_delete	Releases allocated data.

?trnlsp_init

Initializes the solver of a nonlinear least squares problem.

Syntax

```
res = strnlsp_init(handle, n, m, x, eps, iter1, iter2, rs)
res = dtrnlsp_init(handle, n, m, x, eps, iter1, iter2, rs)
```

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?trnlsp_init routine initializes the solver.

After initialization, all subsequent invocations of the ?trnlsp_solve routine should use the values of the handle returned by ?trnlsp_init. This handle stores internal data, including pointers to the arrays *x* and *eps*. It is important to not move or deallocate these arrays until after calling the ?trnlsp_delete routine.

The *eps* array contains a number indicating the stopping criteria:

eps Value	Description
1	$\Delta < eps(1)$
2	$\|\|F(x)\|\|_2 < eps(2)$
3	The Jacobian matrix is singular. $\|\|J(x)_{(1:m,j)}\|\|_2 < eps(3),\ j = 1, \ldots, n$
4	$\|\|s\|\|_2 < eps(4)$
5	$\|\|F(x)\|\|_2 - \|\|F(x) - J(x)s\|\|_2 < eps(5)$
6	The trial step precision. If $eps(6) = 0$, then the trial step meets the required precision ($\leq 1.0*10^{-10}$).

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.

- `s` is the trial step.

Input Parameters

n	`INTEGER`. Length of *x*.
m	`INTEGER`. Length of *F*(*x*).

x	`REAL` for `strnlsp_init`
	`DOUBLE PRECISION` for `dtrnlsp_init`
	Array of size *n*. Initial guess. A reference to this array is stored in *handle* for later use and modification by `?trnlsp_solve`
	.

eps	`REAL` for `strnlsp_init`
	`DOUBLE PRECISION` for `dtrnlsp_init`
	Array of size 6; contains stopping criteria. See the values in the Description section.
	A reference to this array is stored in *handle* for later use by `?trnlsp_solve`
	.

iter1	`INTEGER`. Specifies the maximum number of iterations.
iter2	`INTEGER`. Specifies the maximum number of iterations of trial step calculation.

rs	`REAL` for `strnlsp_init`
	`DOUBLE PRECISION` for `dtrnlsp_init`
	Definition of initial size of the trust region (boundary of the trial step). The recommend minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set *rs* to 0.0, the solver uses the default value, which is 100.0.

Output Parameters

handle	Type `INTEGER*8`.
res	`INTEGER`. Indicates task completion status. • *res* = `TR_SUCCESS` - the routine completed the task normally. • *res* = `TR_INVALID_OPTION` - there was an error in the input parameters. • *res* = `TR_OUT_OF_MEMORY` - there was a memory error. `TR_SUCCESS`, `TR_INVALID_OPTION`, and `TR_OUT_OF_MEMORY` are defined in the `mkl_rci.fi` include file.

See Also

`?trnlsp_solve`

?trnlsp_check

Checks the correctness of handle and arrays containing Jacobian matrix, objective function, and stopping criteria.

Syntax

`res = strnlsp_check(handle, n, m, fjac, fvec, eps, info)`

`res = dtrnlsp_check(handle, n, m, fjac, fvec, eps, info)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `?trnlsp_check` routine checks the arrays passed into the solver as input parameters. If an array contains any `INF` or `NaN` values, the routine sets the flag in output array *info*(see the description of the values returned in the Output Parameters section for the `info` array).

Input Parameters

handle	Type `INTEGER*8`.
n	`INTEGER`. Length of *x*.
m	`INTEGER`. Length of *F*(*x*).

fjac	`REAL` for `strnlsp_check`
	`DOUBLE PRECISION` for `dtrnlsp_check`
	Array of size *m* by *n*. Contains the Jacobian matrix of the function.

fvec	`REAL` for `strnlsp_check`
	`DOUBLE PRECISION` for `dtrnlsp_check`
	Array of size *m*. Contains the function values at `X`, where `fvec(i) = (`$y_i - f_i(x)$`)`.

eps	`REAL` for `strnlsp_check`
	`DOUBLE PRECISION` for `dtrnlsp_check`
	Array of size 6; contains stopping criteria. See the values in the Description section of the `?trnlsp_init`.

Output Parameters

info	`INTEGER` Array of size 6. Results of input parameter checking:

Parameter	Used for	Value	Description
info(1)	Flags for *handle*	0	The handle is valid.
		1	The handle is not allocated.
info(2)	Flags for *fjac*	0	The *fjac* array is valid.
		1	The *fjac* array is not allocated
		2	The *fjac* array contains NaN.
		3	The *fjac* array contains Inf.
info(3)	Flags for *fvec*	0	The *fvec* array is valid.
		1	The *fvec* array is not allocated
		2	The *fvec* array contains NaN.
		3	The *fvec* array contains Inf.
info(4)	Flags for *eps*	0	The *eps* array is valid.
		1	The *eps* array is not allocated
		2	The *eps* array contains NaN.
		3	The *eps* array contains Inf.
		4	The *eps* array contains a value less than or equal to zero.

res INTEGER. Information about completion of the task.

res = TR_SUCCESS - the routine completed the task normally.

TR_SUCCESS is defined in the mkl_rci.fi include file.

?trnlsp_solve

Solves a nonlinear least squares problem using the TR algorithm.

Syntax

res = strnlsp_solve(*handle*, *fvec*, *fjac*, *RCI_Request*)

res = dtrnlsp_solve(*handle*, *fvec*, *fjac*, *RCI_Request*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?trnlsp_solve routine uses the TR algorithm to solve nonlinear least squares problems.

The problem is stated as follows:

$$\min_{x \in R^n} \left\| F(x) \right\|_2^2 = \min_{x \in R^n} \left\| y - f(x) \right\|_2^2, \; y \in R^m, \; x \in R^n, \; f : R^n \to R^m, \; m \geq n,$$

where

- $F(x) : R^n \to R^m$
- $m \geq n$

From a current point $x_{current}$, the algorithm uses the trust-region approach:

$$\min_{x \in R^n} \left\| F(x_{current}) + J(x_{current})(x_{new} - x_{current}) \right\|_2^2 \quad \text{subject to } \left\| x_{new} - x_{current} \right\| \leq \Delta_{current}$$

to get $x_{new} = x_{current} + s$ that satisfies

$$\min_{x \in R^n} \left\| J^T(x) J(x) s + J^T F(x) \right\|_2^2$$

where

- $J(x)$ is the Jacobian matrix
- s is the trial step
- $||s||_2 \leq \Delta_{current}$
- Δ is the trust-region area.

The *RCI_Request* parameter provides additional information:

RCI_Request **Value**	**Description**
2	Request to calculate the Jacobian matrix and put the result into *fjac*
1	Request to recalculate the function at vector X and put the result into *fvec*
0	One successful iteration step on the current trust-region radius (that does not mean that the value of *x* has changed)
-1	The algorithm has exceeded the maximum number of iterations
-2	$\Delta < eps(1)$
-3	$\|\|F(x)\|\|_2 < eps(2)$
-4	The Jacobian matrix is singular. $\|\|J(x)_{(1:m,j)}\|\|_2 < eps(3), \; j = 1, \ldots, n$
-5	$\|\|s\|\|_2 < eps(4)$
-6	$\|\|F(x)\|\|_2 - \|\|F(x) - J(x)s\|\|_2 < eps(5)$

NOTE
If it is possible to combine computations of the function and the jacobian (`RCI_Request = 1` and `2`), you can do that and provide both updated values for *fvec* and *fjac* as fulfillment of `RCI_Request =1` (and do nothing for `RCI_Request = 2`).

Input Parameters

handle Type `INTEGER*8`.

fvec `REAL` for `strnlsp_solve`

`DOUBLE PRECISION` for `dtrnlsp_solve`

Array of size *m*. Contains the function values at `X`, where *fvec*`(i) = (`y_i – $f_i(x)$`)`.

fjac `REAL` for `strnlsp_solve`

`DOUBLE PRECISION` for `dtrnlsp_solve`

Array of size *m* by *n*. Contains the Jacobian matrix of the function.

Output Parameters

fvec `REAL` for `strnlsp_solve`

`DOUBLE PRECISION` for `dtrnlsp_solve`

Array of size *m*. Updated function evaluated at `x`.

RCI_Request `INTEGER`. Informs about the task stage.

See the Description section for the parameter values and their meaning.

res `INTEGER`. Indicates the task completion.

res = `TR_SUCCESS` - the routine completed the task normally.

`TR_SUCCESS` is defined in the `mkl_rci.fi` include file.

?trnlsp_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

res = `strnlsp_get(`*handle*`, `*iter*`, `*st_cr*`, `*r1*`, `*r2*`)`

res = `dtrnlsp_get(`*handle*`, `*iter*`, `*st_cr*`, `*r1*`, `*r2*`)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The initial residual is the value of the functional `(||`*y* – *f*`(`*x*`)||)` of the initial *x* values provided by the user.

The final residual is the value of the functional $(||y - f(x)||)$ of the final *x* resulting from the algorithm operation.

The *st_cr* parameter contains a number indicating the stop criterion:

st_cr **Value**	**Description**
1	The algorithm has exceeded the maximum number of iterations
2	$\Delta < eps(1)$
3	$\|\|F(x)\|\|_2 < eps(2)$
4	The Jacobian matrix is singular. $\|\|J(x)_{(1:m,j)}\|\|_2 < eps(3),\ j = 1, \ldots, n$
5	$\|\|s\|\|_2 < eps(4)$
6	$\|\|F(x)\|\|_2 - \|\|F(x) - J(x)s\|\|_2 < eps(5)$

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

handle Type INTEGER*8.

Output Parameters

iter INTEGER. Contains the current number of iterations.

st_cr INTEGER. Contains the stop criterion.

See the Description section for the parameter values and their meanings.

r1 REAL for strnlsp_get

DOUBLE PRECISION for dtrnlsp_get

Contains the residual, $(||y - f(x)||)$ given the initial *x*.

r2 REAL for strnlsp_get

DOUBLE PRECISION for dtrnlsp_get

Contains the final residual, that is, the value of the functional $(||y - f(x)||)$ of the final *x* resulting from the algorithm operation.

res INTEGER. Indicates the task completion.

res = TR_SUCCESS - the routine completed the task normally.

TR_SUCCESS is defined in the mkl_rci.fi include file.

?trnlsp_delete

Releases allocated data.

Syntax

res = strnlsp_delete(*handle*)

res = dtrnlsp_delete(*handle*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?trnlsp_delete routine releases all memory allocated for the handle. Only after calling this routine is it safe for the user to move or deallocate the memory referenced by *x* and *eps*.

This routine flags memory as not used, but to actually release all memory you must call the support function mkl_free_buffers.

Input Parameters

handle	Type INTEGER*8.

Output Parameters

res	INTEGER. Indicates the task completion. *res* = TR_SUCCESS means the routine completed the task normally. TR_SUCCESS is defined in the mkl_rci.fi include file.

Nonlinear Least Squares Problem with Linear (Bound) Constraints

The nonlinear least squares problem with linear bound constraints is very similar to the nonlinear least squares problem without constraints but it has the following constraints:

$$l_i \le x_i \le u_i, i = 1, \ldots, n, \quad l, u \in R^n.$$

See usage examples in the examples\f\nonlinear_solvers folderof your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see ex_nlsqp_bc_f.f .

NOTE There are two options for handling the boundary constraints enabled through the parameter array, see the description of eps(5)

RCI TR Routines for Problem with Bound Constraints

Routine Name	Operation
?trnlspbc_init	Initializes the solver.
?trnlspbc_check	Checks correctness of the input parameters.
?trnlspbc_solve	Solves a nonlinear least squares problem using RCI and the Trust-Region algorithm.
?trnlspbc_get	Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Routine Name	Operation
?trnlspbc_delete	Releases allocated data.

?trnlspbc_init

Initializes the solver of nonlinear least squares problem with linear (boundary) constraints.

Syntax

res = strnlspbc_init(*handle*, *n*, *m*, *x*, *LW*, *UP*, *eps*, *iter1*, *iter2*, *rs*)

res = dtrnlspbc_init(*handle*, *n*, *m*, *x*, *LW*, *UP*, *eps*, *iter1*, *iter2*, *rs*)

Description

The ?trnlspbc_init routine initializes the solver.

After initialization, all subsequent invocations of the ?trnlspbc_solve routine should use the values of the handle returned by ?trnlspbc_init. This handle stores internal data, including pointers to the arrays *x*, *LW*, *UP*, and *eps*. It is important to not move or deallocate these arrays until after calling the ?trnlspbc_delete routine.

The *eps* array contains a number indicating the stopping criteria:

eps Value	Description
1	$\Delta < eps(1)$
2	$\|\|F(x)\|\|_2 < eps(2)$
3	The Jacobian matrix is singular. $\|\|J(x)_{(1:m,j)}\|\|_2 < eps(3),\ j = 1, \ldots, n$
4	$\|\|s\|\|_2 < eps(4)$
5	$\|\|F(x)\|\|_2 - \|\|F(x) - J(x)s\|\|_2 < \|eps(5)\|$ **NOTE** If eps(5) > 0, an extra scaling is applied to 's' after it has been selected, to ensure that it does not leave the specified domain, but scales it down to not cross the boundary. This preserves the solution inside the boundary, but may result in getting stuck in a local minimum on the boundary and exiting early due to this stopping criteria If eps(5) < 0, extra scaling is not applied, which may result in the solution, x, leaving the domain. If this occurs, try starting over with a different initial condition.
6	The trial step precision. If $eps(6) = 0$, then the trial step meets the required precision ($\leq 1.0*10^{-10}$).

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.

- `s` is the trial step.

Input Parameters

n	`INTEGER`. Length of *x*.
m	`INTEGER`. Length of *F*(*x*).

x	`REAL` for `strnlspbc_init`
	`DOUBLE PRECISION` for `dtrnlspbc_init`
	Array of size *n*. Initial guess. A reference to this array is stored in handle for later use and modification by `?trnlspbc_solve`.

LW	`REAL` for `strnlspbc_init`
	`DOUBLE PRECISION` for `dtrnlspbc_init`
	Array of size *n*.
	Contains low bounds for *x* ($lw_i < x_i$). A reference to this array is stored in handle for later use by `?trnlspbc_solve`.

UP	`REAL` for `strnlspbc_init`
	`DOUBLE PRECISION` for `dtrnlspbc_init`
	Array of size *n*.
	Contains upper bounds for *x* ($up_i > x_i$). A reference to this array is stored in handle for later use by `?trnlspbc_solve`.

eps	`REAL` for `strnlspbc_init`
	`DOUBLE PRECISION` for `dtrnlspbc_init`
	Array of size 6; contains stopping criteria. See the values in the Description section. A reference to this array is stored in handle for later use by `?trnlspbc_solve`.

iter1	`INTEGER`. Specifies the maximum number of iterations.
iter2	`INTEGER`. Specifies the maximum number of iterations of trial step calculation.

rs	`REAL` for `strnlspbc_init`
	`DOUBLE PRECISION` for `dtrnlspbc_init`
	Definition of initial size of the trust region (boundary of the trial step). The recommended minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set *rs* to 0.0, the solver uses the default value, which is 100.0.

Output Parameters

handle	Type `INTEGER*8`.
res	`INTEGER`. Informs about the task completion.

- *res* = TR_SUCCESS - the routine completed the task normally.
- *res* = TR_INVALID_OPTION - there was an error in the input parameters.
- *res* = TR_OUT_OF_MEMORY - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.fi include file.

?trnlspbc_check

Checks the correctness of handle and arrays containing Jacobian matrix, objective function, lower and upper bounds, and stopping criteria.

Syntax

res = strnlspbc_check(*handle*, *n*, *m*, *fjac*, *fvec*, *LW*, *UP*, *eps*, *info*)

res = dtrnlspbc_check(*handle*, *n*, *m*, *fjac*, *fvec*, *LW*, *UP*, *eps*, *info*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?trnlspbc_check routine checks the arrays passed into the solver as input parameters. If an array contains any INF or NaN values, the routine sets the flag in output array *info*(see the description of the values returned in the Output Parameters section for the *info* array).

Input Parameters

handle
Type INTEGER*8.

n
INTEGER. Length of *x*.

m
INTEGER. Length of *F*(*x*).

fjac
REAL for strnlspbc_check
DOUBLE PRECISION for dtrnlspbc_check
Array of size *m* by *n*. Contains the Jacobian matrix of the function.

fvec
REAL for strnlspbc_check
DOUBLE PRECISION for dtrnlspbc_check
Array of size *m*. Contains the function values at X, where *fvec*(*i*) = (y_i – $f_i(x)$).

LW
REAL for strnlspbc_check
DOUBLE PRECISION for dtrnlspbc_check
Array of size *n*.
Contains low bounds for *x* ($lw_i < x_i$).

UP
REAL for strnlspbc_check
DOUBLE PRECISION for dtrnlspbc_check
Array of size *n*.

Contains upper bounds for x ($up_i > x_i$).

eps

`REAL` for `strnlspbc_check`

`DOUBLE PRECISION` for `dtrnlspbc_check`

Array of size 6; contains stopping criteria. See the values in the Description section of the `?trnlspbc_init`.

Output Parameters

info

`INTEGER`

Array of size 6.

Results of input parameter checking:

Parameter	Used for	Value	Description
info(1)	Flags for *handle*	0	The handle is valid.
		1	The handle is not allocated.
info(2)	Flags for *fjac*	0	The *fjac* array is valid.
		1	The *fjac* array is not allocated
		2	The *fjac* array contains `NaN`.
		3	The *fjac* array contains `Inf`.
info(3)	Flags for *fvec*	0	The *fvec* array is valid.
		1	The *fvec* array is not allocated
		2	The *fvec* array contains `NaN`.
		3	The *fvec* array contains `Inf`.
info(4)	Flags for *LW*	0	The *LW* array is valid.
		1	The *LW* array is not allocated
		2	The *LW* array contains `NaN`.
		3	The *LW* array contains `Inf`.
		4	The lower bound is greater than the upper bound.
info(5)	Flags for *up*	0	The *up* array is valid.
		1	The *up* array is not allocated
		2	The *up* array contains `NaN`.
		3	The *up* array contains `Inf`.

Parameter	Used for	Value	Description
		4	The upper bound is less than the lower bound.
info(6)	Flags for *eps*	0	The *eps* array is valid.
		1	The *eps* array is not allocated
		2	The *eps* array contains NaN.
		3	The *eps* array contains Inf.
		4	The *eps* array contains a value less than or equal to zero.

res INTEGER. Information about completion of the task.

res = TR_SUCCESS - the routine completed the task normally.

TR_SUCCESS is defined in the mkl_rci.fi include file.

?trnlspbc_solve

Solves a nonlinear least squares problem with linear (bound) constraints using the Trust-Region algorithm.

Syntax

res = strnlspbc_solve(*handle*, *fvec*, *fjac*, *RCI_Request*)

res = dtrnlspbc_solve(*handle*, *fvec*, *fjac*, *RCI_Request*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?trnlspbc_solve routine, based on RCI, uses the Trust-Region algorithm to solve nonlinear least squares problems with linear (bound) constraints. The problem is stated as follows:

$$\min_{x \in R^n} \left\| F(x) \right\|_2^2 = \min_{x \in R^n} \left\| y - f(x) \right\|_2^2, \; y \in R^m, \; x \in R^n, \; f : R^n \to R^m, \; m \geq n$$

where

$l_i \leq x_i \leq u_i$

$i = 1, \ldots, n$.

The *RCI_Request* parameter provides additional information:

RCI_Request Value	Description
2	Request to calculate the Jacobian matrix and put the result into *fjac*
1	Request to recalculate the function at vector X and put the result into *fvec*

RCI_Request Value	Description
0	One successful iteration step on the current trust-region radius (that does not mean that the value of *x* has changed)
-1	The algorithm has exceeded the maximum number of iterations
-2	$\Delta < eps(1)$
-3	$\|\|F(x)\|\|_2 < eps(2)$
-4	The Jacobian matrix is singular. $\|\|J(x)_{(1:m,j)}\|\|_2 < eps(3),\ j = 1, \ldots, n$
-5	$\|\|s\|\|_2 < eps(4)$
-6	$\|\|F(x)\|\|_2 - \|\|F(x) - J(x)s\|\|_2 < \|eps(5)\|$

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- `s` is the trial step.

Input Parameters

handle
Type `INTEGER*8`.

fvec
`REAL` for `strnlspbc_solve`
`DOUBLE PRECISION` for `dtrnlspbc_solve`
Array of size *m*. Contains the function values at `X`, where $fvec(i) = (y_i - f_i(x))$.

fjac
`REAL` for `strnlspbc_solve`
`DOUBLE PRECISION` for `dtrnlspbc_solve`
Array of size *m* by *n*. Contains the Jacobian matrix of the function.

Output Parameters

fvec
`REAL` for `strnlspbc_solve`
`DOUBLE PRECISION` for `dtrnlspbc_solve`
Array of size *m*. Updated function evaluated at `x`.

RCI_Request
`INTEGER`. Informs about the task stage.
See the Description section for the parameter values and their meaning.

res
`INTEGER`. Informs about the task completion.
res = `TR_SUCCESS` means the routine completed the task normally.
`TR_SUCCESS` is defined in the `mkl_rci.fi` include file.

?trnlspbc_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

`res = strnlspbc_get(handle, iter, st_cr, r1, r2)`

`res = dtrnlspbc_get(handle, iter, st_cr, r1, r2)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The *st_cr* parameter contains a number indicating the stop criterion:

st_cr **Value**	**Description**
1	The algorithm has exceeded the maximum number of iterations
2	$\Delta < eps(1)$
3	$\|\|F(x)\|\|_2 < eps(2)$
4	The Jacobian matrix is singular. $\|\|J(x)_{(1:m,j)}\|\|_2 < eps(3),\ j = 1, \ldots, n$
5	$\|\|s\|\|_2 < eps(4)$
6	$\|\|F(x)\|\|_2 - \|\|F(x) - J(x)s\|\|_2 < eps(5)$

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

handle Type `INTEGER*8`.

Output Parameters

iter `INTEGER`. Contains the current number of iterations.

st_cr `INTEGER`. Contains the stop criterion.

See the Description section for the parameter values and their meanings.

r1 `REAL` for `strnlspbc_get`

`DOUBLE PRECISION` for `dtrnlspbc_get`

Contains the residual, $(||y - f(x)||)$ given the initial x.

r2	REAL for strnlspbc_get DOUBLE PRECISION for dtrnlspbc_get Contains the final residual, that is, the value of the function (||*y* – *f*(*x*)||) of the final *x* resulting from the algorithm operation.
res	INTEGER. Informs about the task completion. *res* = TR_SUCCESS - the routine completed the task normally. TR_SUCCESS is defined in the mkl_rci.fi include file.

?trnlspbc_delete

Releases allocated data.

Syntax

res = strnlspbc_delete(*handle*)

res = dtrnlspbc_delete(*handle*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?trnlspbc_delete routine releases all memory allocated for the handle. Only after calling this routine is it safe for the user to move or deallocate the memory referenced by *x, LW, UP,* and *eps.*

> **NOTE**
> This routine flags memory as not used, but to actually release all memory you must call the support function mkl_free_buffers.

Input Parameters

handle	Type INTEGER*8.

Output Parameters

res	INTEGER. Informs about the task completion. *res* = TR_SUCCESS means the routine completed the task normally. TR_SUCCESS is defined in the mkl_rci.fi include file.

Jacobian Matrix Calculation Routines

This section describes routines that compute the Jacobian matrix using the central difference algorithm. Jacobian matrix calculation is required to solve a nonlinear least squares problem and systems of nonlinear equations (with or without linear bound constraints). Routines for calculation of the Jacobian matrix have the "Black-Box" interfaces, where you pass the objective function via parameters. Your objective function must have a fixed interface.

Jacobian Matrix Calculation Routines

Routine Name	Operation
?jacobi_init	Initializes the solver.
?jacobi_solve	Computes the Jacobian matrix of the function on the basis of RCI using the central difference algorithm.
?jacobi_delete	Removes data.
?jacobi	Computes the Jacobian matrix of the fcn function using the central difference algorithm.
?jacobix	Presents an alternative interface for the ?jacobi function enabling you to pass additional data into the objective function.

?jacobi_init

Initializes the solver for Jacobian calculations.

Syntax

res = sjacobi_init(*handle*, *n*, *m*, *x*, *fjac*, *eps*)

res = djacobi_init(*handle*, *n*, *m*, *x*, *fjac*, *eps*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The routine initializes the solver.

Input Parameters

n

INTEGER. Length of *x*.

m

INTEGER. Length of *F*.

x

REAL for sjacobi_init

DOUBLE PRECISION for djacobi_init

Array of size *n*. Vector, at which the function is evaluated.

A reference to this array is stored in *handle* for later use and modification by ?jacobi_solve.

eps

REAL for sjacobi_init

DOUBLE PRECISION for djacobi_init

Precision of the Jacobian matrix calculation.

fjac

REAL for sjacobi_init

DOUBLE PRECISION for djacobi_init

Array of size *m* by *n*. Contains the Jacobian matrix of the function.

A reference to this array is stored in *handle* for later use and modification by ?jacobi_solve.

Output Parameters

handle	Data object of the INTEGER*8. Stores internal data, including pointers to the user-provided arrays *x* and *fjac*. It is important that the user does not move or deallocate these arrays until after calling the ?jacobi_delete routine.
res	INTEGER. Indicates task completion status. • *res* = TR_SUCCESS - the routine completed the task normally. • *res* = TR_INVALID_OPTION - there was an error in the input parameters. • *res* = TR_OUT_OF_MEMORY - there was a memory error. TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.fi include file.

?jacobi_solve

Computes the Jacobian matrix of the function using RCI and the central difference algorithm.

Syntax

res = sjacobi_solve(*handle*, *f1*, *f2*, *RCI_Request*)

res = djacobi_solve(*handle*, *f1*, *f2*, *RCI_Request*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?jacobi_solve routine computes the Jacobian matrix of the function using RCI and the central difference algorothm.

See usage examples in the examples\solverf\source folderof your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see sjacobi_rci_f.f and djacobi_rci_f.f.

Input Parameters

handle	Type INTEGER*8.
RCI_Request	INTEGER. Set to 0 before the first call to ?jacobi_solve.

Output Parameters

f1	REAL for sjacobi_solve DOUBLE PRECISION for djacobi_solve Contains the updated function values at x + *eps*.
f2	REAL for sjacobi_solve DOUBLE PRECISION for djacobi_solve Array of size *m*. Contains the updated function values at x - *eps*.
RCI_Request	Provides information about the task completion. When equal to 0, the task has completed successfully.

	RCI_Request= 1 indicates that you should compute the function values at the current *x* point and put the results into *f1*. *RCI_Request*= 2 indicates that you should compute the function values at the current *x* point and put the results into *f2*.
res	INTEGER. Indicates the task completion status. • *res* = TR_SUCCESS - the routine completed the task normally. • *res* = TR_INVALID_OPTION - there was an error in the input parameters. TR_SUCCESS and TR_INVALID_OPTION are defined in the mkl_rci.fi include file.

See Also

?jacobi_init

?jacobi_delete

Releases allocated data.

Syntax

res = sjacobi_delete(*handle*)

res = djacobi_delete(*handle*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?jacobi_delete routine releases all memory allocated for the handle. Only after calling this routine is it safe for the user to move or deallocate the memory referenced by *x* and *fjac*.

This routine flags memory as not used, but to actually release all memory you must call the support function mkl_free_buffers.

Input Parameters

handle	Type INTEGER*8.

Output Parameters

res	INTEGER. Informs about the task completion. *res* = TR_SUCCESS means the routine completed the task normally. TR_SUCCESS is defined in the mkl_rci.fi include file.

?jacobi

Computes the Jacobian matrix of the objective function using the central difference algorithm.

Syntax

res = sjacobi(*fcn*, *n*, *m*, *fjac*, *x*, *eps*)

res = djacobi(*fcn*, *n*, *m*, *fjac*, *x*, *eps*)

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `?jacobi` routine computes the Jacobian matrix for function *fcn* using the central difference algorithm. This routine has a "Black-Box" interface, where you input the objective function via parameters. Your objective function must have a fixed interface.

See calling and usage examples in the `examples\solverf\source` folderof your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see `ex_nlsqp_f.f` and `ex_nlsqp_bc_f.f` .

Input Parameters

fcn

User-supplied subroutine to evaluate the function that defines the least squares problem. Called as `fcn` (*m*, *n*, *x*, *f*) with the following parameters:

Parameter	Type	Description
Input Parameters		
m	`INTEGER`	Length of *f*.
n	`INTEGER`	Length of *x*.
x	`REAL` for `sjacobi` `DOUBLE PRECISION` for `djacobi`	Array of size *n*. Vector, at which the function is evaluated. The `fcn` function should not change this parameter.
Output Parameters		
f	`REAL` for `sjacobix` `DOUBLE PRECISION` for `djacobix`	Array of size *m*; contains the function values at *x*.

You need to declare `fcn` as `EXTERNAL` in the calling program.

n

`INTEGER`. Length of *X*.

m

`INTEGER`. Length of *F*.

x

`REAL` for `sjacobi`

`DOUBLE PRECISION` for `djacobi`

Array of size *n*. Vector at which the function is evaluated.

eps

`REAL` for `sjacobi`

`DOUBLE PRECISION` for `djacobi`

Precision of the Jacobian matrix calculation.

Output Parameters

fjac

`REAL` for `sjacobi`

`DOUBLE PRECISION` for `djacobi`

Array of size *m* by *n*. Contains the Jacobian matrix of the function.

res INTEGER. Indicates task completion status.

- *res* = TR_SUCCESS - the routine completed the task normally.
- *res* = TR_INVALID_OPTION - there was an error in the input parameters.
- *res* = TR_OUT_OF_MEMORY - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.fi include file.

See Also

?jacobix

?jacobix

Alternative interface for?jacobi *function for passing additional data into the objective function.*

Syntax

res = sjacobix(*fcn*, *n*, *m*, *fjac*, *x*, *eps*, *user_data*)

res = djacobix(*fcn*, *n*, *m*, *fjac*, *x*, *eps*, *user_data*)

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?jacobix routine presents an alternative interface for the ?jacobi function that enables you to pass additional data into the objective function *fcn*.

See calling and usage examples in the examples\solverf\source folderof your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see ex_nlsqp_f90_x.f90 and ex_nlsqp_bc_f90_x.f90.

Input Parameters

fcn User-supplied subroutine to evaluate the function that defines the least squares problem. Called as fcn (*m*, *n*, *x*, *f*, *user_data*) with the following parameters:

Parameter	Type	Description
Input Parameters		
m	INTEGER	Length of *f*.
n	INTEGER	Length of *x*.
x	REAL for sjacobix DOUBLE PRECISION for djacobix	Array of size *n*. Vector, at which the function is evaluated. The fcn function should not change this parameter.
user_data	INTEGER(C_INTPTR_T), for Fortran	Reference to your additional data, if any, passed by value: *user_data*=%VAL(LOC(*data*)). Otherwise, a dummy argument.

Parameter	Type	Description
Output Parameters		
f	REAL for sjacobix DOUBLE PRECISION for djacobix	Array of size *m*; contains the function values at *x*.

You need to declare fcn as EXTERNAL in the calling program.

n INTEGER. Length of *X*.

m INTEGER. Length of *F*.

x REAL for sjacobix

DOUBLE PRECISION for djacobix

Array of size *n*. Vector at which the function is evaluated.

eps REAL for sjacobix

DOUBLE PRECISION for djacobix

Precision of the Jacobian matrix calculation.

user_data INTEGER(C_INTPTR_T). Reference to your additional data, passed by value: *user_data*=%VAL(LOC(*data*)). Otherwise, a dummy argument.

Output Parameters

fjac REAL for sjacobix

DOUBLE PRECISION for djacobix

Array of size *m* by *n*). Contains the Jacobian matrix of the function.

res INTEGER. Indicates task completion status.

- *res* = TR_SUCCESS - the routine completed the task normally.
- *res* = TR_INVALID_OPTION - there was an error in the input parameters.
- *res* = TR_OUT_OF_MEMORY - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.fi include file.

See Also

?jacobi

Support Functions

Intel® oneAPI Math Kernel Library (oneMKL) support functions are subdivided into the following groups according to their purpose:

Version Information

Threading Control

Error Handling

Character Equality Testing

The following table lists Intel® oneAPI Math Kernel Library (oneMKL) support functions.

oneMKL Support Functions

Function Name	Operation
Version Information	
mkl_get_version_string	Returns the Intel® oneAPI Math Kernel Library (oneMKL) version in a character string.
Threading Control	
mkl_set_num_threads	Specifies the number of OpenMP* threads to use.
mkl_domain_set_num_threads	Specifies the number of OpenMP* threads for a particular function domain.
mkl_set_num_threads_local	Specifies the number of OpenMP* threads for all Intel® oneAPI Math Kernel Library (oneMKL) functions on the current execution thread.
mkl_set_dynamic	Enables Intel® oneAPI Math Kernel Library (oneMKL) to dynamically change the number of OpenMP* threads.
mkl_get_max_threads	Gets the number of OpenMP* threads targeted for parallelism.
mkl_domain_get_max_threads	Gets the number of OpenMP* threads targeted for parallelism for a particular function domain.
mkl_get_dynamic	Determines whether Intel® oneAPI Math Kernel Library (oneMKL) is enabled to dynamically change the number of OpenMP* threads.
mkl_set_num_stripes	Specifies the number of partitions along the leading dimension of the output matrix for parallel ?GEMM functions.
mkl_get_num_stripes	Gets the number of partitions along the leading dimension of the output matrix for parallel ?GEMM functions.
Error Handling	
xerbla	Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.
pxerbla	Handles error conditions for the ScaLAPACK routines.
mkl_set_exit_handler	Sets the custom handler of fatal errors.
Character Equality Testing	
lsame	Tests two characters for equality regardless of the case.
lsamen	Tests two character strings for equality regardless of the case.

Function Name	Operation
Timing	
second/dsecnd	Returns elapsed time in seconds. Use to estimate real time between two calls to this function.
mkl_get_cpu_clocks	Returns elapsed CPU clocks.
mkl_get_cpu_frequency	Returns CPU frequency value in GHz.
mkl_get_max_cpu_frequency	Returns the maximum CPU frequency value in GHz.
mkl_get_clocks_frequency	Returns the frequency value in GHz based on constant-rate Time Stamp Counter.
Memory Management	
mkl_free_buffers	Frees unused memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.
mkl_thread_free_buffers	Frees unused memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator in the current thread.
mkl_mem_stat	Reports the status of the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.
mkl_peak_mem_usage	Reports the peak memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.
mkl_disable_fast_mm	Turns off the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator for Intel® oneAPI Math Kernel Library (oneMKL) functions to directly use the system`malloc`/`free` functions.
mkl_malloc	Allocates an aligned memory buffer.
mkl_calloc	Allocates and initializes an aligned memory buffer.
mkl_realloc	Changes the size of memory buffer allocated by `mkl_malloc`/`mkl_calloc`.
mkl_free	Frees the aligned memory buffer allocated by `mkl_malloc`/`mkl_calloc`.
mkl_set_memory_limit	On Linux, sets the limit of memory that Intel® oneAPI Math Kernel Library (oneMKL) can allocate for a specified type of memory.
Single Dynamic Library (SDL) Control	
mkl_set_interface_layer	Sets the interface layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time.
mkl_set_threading_layer	Sets the threading layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time.
mkl_set_xerbla	Replaces the error handling routine. Use with the Single Dynamic Library .
mkl_set_progress	Replaces the progress information routine.

Function Name	Operation
mkl_set_pardiso_pivot	Replaces the routine handling Intel® oneAPI Math Kernel Library (oneMKL) PARDISO pivots with a user-defined routine. Use with the Single Dynamic Library (SDL).
Conditional Numerical Reproducibility (CNR) Control	
mkl_cbwr_set	Configures the CNR mode of Intel® oneAPI Math Kernel Library (oneMKL).
mkl_cbwr_get	Returns the current CNR settings.
mkl_cbwr_get_auto_branch	Automatically detects the CNR code branch for your platform.
Miscellaneous	
mkl_progress	Provides progress information.
mkl_enable_instructions	
mkl_set_env_mode	Set up the mode that ignores environment settings specific to Intel® oneAPI Math Kernel Library (oneMKL).
mkl_verbose	Enable or disable Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode.
mkl_verbose_output_file	Write output in Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode to a file.
mkl_set_mpi	Sets the implementation of the message-passing interface to be used by Intel® oneAPI Math Kernel Library (oneMKL).
mkl_finalize	Terminates Intel® oneAPI Math Kernel Library (oneMKL) execution environment and frees resources allocated by the library.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Using a Fortran Interface Module for Support Functions

To call a support function from your Fortran application, include one of the following statements in your code:

- `INCLUDE mkl_service.fi` or `INCLUDE mkl.fi`
- `USE mkl_service`

The `USE` statement references the `mkl_service.mod`interface module corresponding to your architecture and programming interface. The module provides an application programming interface to Intel® oneAPI Math Kernel Library (oneMKL) support entities, such as subroutines and constants. Because Fortran interface modules are compiler-dependent, Intel® oneAPI Math Kernel Library (oneMKL) offers the`mkl_service.f90` source file for the module, as well as architecture-specific and interface-specific `mkl_service`modules precompiled with the Intel® Fortran or Intel® Visual Fortran compiler. These modules are available in the following subdirectories of the Intel® oneAPI Math Kernel Library (oneMKL) include directory:

Architecture, Interface	Subdirectory of the Intel® oneAPI Math Kernel Library (oneMKL) Installation Directory
IA-32	`include\ia32`
Intel® 64, LP64	`include\intel64\lp64`
Intel® 64, ILP64	`include\intel64\ilp64`

To ensure that your application searches the right module, specify the appropriate subdirectory during compilation as an additional directory for the include path (through the `/I` option on Windows* OS or the `-I` option on Linux* OS or macOS*).

If you are using a non-Intel Fortran compiler, you need to build the module yourself by compiling the `mkl_service.f90`file, available in the Intel® oneAPI Math Kernel Library (oneMKL) include directory.

For more information on compiler-dependent functions and modules, refer to the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Version Information

Intel® oneAPI Math Kernel Library (oneMKL) providesmethods for extracting information about the library version number, such as:

- using the `mkl_get_version` function to obtain an `MKLVersion` structure that contains the version information

A makefile is also provided to automatically build the examples and output summary files containing the version information for the current library.

mkl_get_version_string

Returns the Intel® oneAPI Math Kernel Library (oneMKL) version in a character string.

Syntax

`call mkl_get_version_string( buf )`

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Output Parameters

Name	Type	Description
buf	`CHARACTER*198`	Source string

Description

The function returns a string that contains the Intel® oneAPI Math Kernel Library (oneMKL) version.

For usage details, see the code example below:

Example

```
program test_mkl_get_version_string

character*198  buf
```

```
call mkl_get_version_string(buf)
write(*,'(a)') buf

end
```

Threading Control

Intel® oneAPI Math Kernel Library (oneMKL) provides functions for OpenMP* threading control, discussed in this section.

> **Important**
> If Intel® oneAPI Math Kernel Library (oneMKL) operates within the Intel® Threading Building Blocks (Intel® TBB) execution environment, the environment variables for OpenMP* threading control, such as`OMP_NUM_THREADS`, and Intel® oneAPI Math Kernel Library (oneMKL) functions discussed in this section have no effect. If the Intel TBB threading technology is used, control the number of threads through the Intel TBB application programming interface. Read the documentation for the `tbb::task_scheduler_init` class at https://www.threadingbuildingblocks.org/docs/doxygen/a00150.html to find out how to specify the number of Intel TBB threads.

If Intel® oneAPI Math Kernel Library (oneMKL) operates within an OpenMP* execution environment, you can control the number of threads for Intel® oneAPI Math Kernel Library (oneMKL) using OpenMP* runtime library routines and environment variables (see the OpenMP* specification for details). Additionally Intel® oneAPI Math Kernel Library (oneMKL) provides*optional*threading control functions and environment variables that enable you to specify the number of threads for Intel® oneAPI Math Kernel Library (oneMKL) and to control dynamic adjustment of the number of threads*independently*of the OpenMP* settings. The settings made with the Intel® oneAPI Math Kernel Library (oneMKL) threading control functions and environment variables do not affect OpenMP* settings but take precedence over them.

If functions are used, Intel® oneAPI Math Kernel Library (oneMKL) environment variables may control Intel® oneAPI Math Kernel Library (oneMKL) threading. For details of those environment variables, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

You can specify the number of threads for Intel® oneAPI Math Kernel Library (oneMKL) function domains with the mkl_set_num_threads or mkl_domain_set_num_threads function. While `mkl_set_num_threads` specifies the number of threads for the entire Intel® oneAPI Math Kernel Library (oneMKL), `mkl_domain_set_num_threads` does it for a specific function domain. The following table lists the function domains that support independent threading control. The table also provides named constants to pass to threading control functions as a parameter that specifies the function domain.

oneMKL Function Domains

Function Domain	Named Constant
Basic Linear Algebra Subroutines (BLAS)	`MKL_DOMAIN_BLAS`
Fast Fourier Transform (FFT) functions, except Cluster FFT functions	`MKL_DOMAIN_FFT`
Vector Math (VM) functions	`MKL_DOMAIN_VML`
Parallel Direct Solver (PARDISO) functions	`MKL_DOMAIN_PARDISO`
All Intel® oneAPI Math Kernel Library (oneMKL) functions except the functions from the domains where the number of threads is set explicitly.	`MKL_DOMAIN_ALL`

Warning
Do not increase the number of OpenMP threads used for cluster_sparse_solver between the first call and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.

Both `mkl_set_num_threads` and `mkl_domain_set_num_threads` functions set the number of threads for all subsequent calls to Intel® oneAPI Math Kernel Library (oneMKL) from all applications threads. Use themkl_set_num_threads_local function to specify different numbers of threads for Intel® oneAPI Math Kernel Library (oneMKL) on different execution threads of your application. The thread-local settings take precedence over the global settings. However, the thread-local settings may have undesirable side effects (see the description of the`mkl_set_num_threads_local` function for details).

By default, Intel® oneAPI Math Kernel Library (oneMKL) canadjust the specified number of threads dynamically. For example, Intel® oneAPI Math Kernel Library (oneMKL) may use fewer threads if the size of the computation is not big enough or not create parallel regions when running within an OpenMP* parallel region. Although Intel® oneAPI Math Kernel Library (oneMKL) may actually use a different number of threads from the number specified, the library does not create parallel regions with more threads than specified. If dynamic adjustment of the number of threads is disabled, Intel® oneAPI Math Kernel Library (oneMKL) attempts to use the specified number of threads in internal parallel regions (for more information, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*). Use the mkl_set_dynamic function to control dynamic adjustment of the number of threads.

mkl_set_num_threads

Specifies the number of OpenMP threads to use.*

Syntax

```
call mkl_set_num_threads( nt )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
nt	INTEGER	*nt* > 0 - The number of threads suggested by the user. *nt*≤ 0 - Invalid value, which is ignored.

Description

This function enables you to specify how many OpenMP threads Intel® oneAPI Math Kernel Library (oneMKL) should use for internal parallel regions. If this number is not set (default), Intel® oneAPI Math Kernel Library (oneMKL) functions use the default number of threads for the OpenMP run-time library. The specified number of threads applies:

- To all Intel® oneAPI Math Kernel Library (oneMKL) functions except the functions from the domains where the number of threads is set withmkl_domain_set_num_threads
- To all execution threads except the threads where the number of threads is set with mkl_set_num_threads_local

The number specified is a hint, and Intel® oneAPI Math Kernel Library (oneMKL) may actually use a smaller number.

NOTE
This function takes precedence over the MKL_NUM_THREADS environment variable.

Example

```
use mkl_service
…
call mkl_set_num_threads(4)
call my_compute_using_mkl  !Intel MKL uses up to 4 OpenMP threads
```

mkl_domain_set_num_threads

Specifies the number of OpenMP threads for a particular function domain.*

Syntax

ierr = mkl_domain_set_num_threads(*nt*, *domain*)

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Input Parameters

Name	Type	Description
nt	INTEGER	*nt* > 0 - The number of threads suggested by the user. *nt* = 0 - The default number of threads for the OpenMP run-time library. *nt* < 0 - Invalid value, which is ignored.
domain	INTEGER	The named constant that defines the targeted domain.

Description

This function specifies how many OpenMP threads a particular function domain of Intel® oneAPI Math Kernel Library (oneMKL) should use. If this number is not set (default) or if it is set to zero in a call to this function, Intel® oneAPI Math Kernel Library (oneMKL) uses the default number of threads for the OpenMP run-time library. The number of threads specified applies to the specified function domain on all execution threads except the threads where the number of threads is set withmkl_set_num_threads_local. For a list of supported values of the *domain* argument, see Table "Intel MKL Function Domains".

The number of threads specified is only a hint, and Intel® oneAPI Math Kernel Library (oneMKL) may actually use a smaller number.

NOTE
This function takes precedence over the MKL_DOMAIN_NUM_THREADS environment variable.

Return Values

Name	Type	Description
ierr	INTEGER	1 - Indicates no error, execution is successful. 0 - Indicates a failure, possibly because of invalid input parameters.

Example

```
use mkl_service
integer(4) :: status
…
status = mkl_domain_set_num_threads(4, MKL_DOMAIN_BLAS)
call my_compute_with_mkl_blas()    !Intel MKL BLAS functions use up to 4 threads
call my_compute_with_mkl_dft()     !Intel MKL FFT functions use the default number of threads
```

mkl_set_num_threads_local

Specifies the number of OpenMP threads for all Intel® oneAPI Math Kernel Library (oneMKL) functions on the current execution thread.*

Syntax

save_nt = mkl_set_num_threads_local(*nt*)

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Input Parameters

Name	Type	Description
nt	INTEGER*4	*nt*> 0 - The number of threads for Intel® oneAPI Math Kernel Library (oneMKL) functions to use on the current execution thread. *nt* = 0 - A request to reset the thread-local number of threads and use the global number.

Description

This function sets the number of OpenMP threads that Intel® oneAPI Math Kernel Library (oneMKL) functions should request for parallel computation. The number of threads is thread-local, which means that it only affects the current execution thread of the application. If the thread-local number is not set or if this number is set to zero in a call to this function, Intel® oneAPI Math Kernel Library (oneMKL) functions use the global number of threads. You can set the global number of threads using themkl_set_num_threads or mkl_domain_set_num_threads function.

The thread-local number of threads takes precedence over the global number: if the thread-local number is non-zero, changes to the global number of threads have no effect on the current thread.

Caution
If your application is threaded with OpenMP* andparallelization of Intel® oneAPI Math Kernel Library (oneMKL) is based on nested OpenMP parallelism,different OpenMP parallel regions reuse OpenMP threads. Therefore a thread-local setting in one OpenMP parallel region may continue to affect not only the master thread after the parallel region ends, but also subsequent parallel regions. To avoid performance implications of this side effect, reset the thread-local number of threads before leaving the OpenMP parallel region (see Examples for how to do it).

Return Values

Name	Type	Description
save_nt	INTEGER*4	The value of the thread-local number of threads that was used before this function call. Zero means that the global number of threads was used.

Examples

This example shows how to avoid the side effect of a thread-local number of threads by reverting to the global setting:

```
use omp_lib
use mkl_service
integer(4) :: dummy
…
call mkl_set_num_threads(16)
call my_compute_using_mkl()    ! Intel MKL functions use up to 16 threads
!$omp parallel num_threads(2)
  if (0 == omp_get_thread(num))  dummy = mkl_set_num_threads_local(4)
  if (1 == omp_get_thread(num))  dummy = mkl_set_num_threads_local(12)
  call my_compute_using_mkl()    ! Intel MKL functions use up to 4 threads on thread 0
                    !   and up to 12 threads on thread 1
!$omp end parallel
call my_compute_using_mkl()    ! Intel MKL functions use up to 4 threads (!)
dummy = mkl_set_num_threads_local(0)    ! make master thread use global setting
call my_compute_using_mkl()    ! Now Intel MKL functions use up to 16 threads
```

This example shows how to avoid the side effect of a thread-local number of threads by saving and restoring the existing setting:

```
subroutine my_compute(nt)
    use mkl_service
    integer(4) :: nt, save
save = mkl_set_num_threads_local( nt ) ! save the Intel® oneAPI Math Kernel Library (oneMKL)
number of threads
    call my_compute_using_mkl()    ! Intel MKL functions use up to nt threads on this thread
save = mkl_set_num_threads_local( save ) ! restore the Intel® oneAPI Math Kernel Library
(oneMKL) number of threads
end subroutine my_compute
```

mkl_set_dynamic

Enables Intel® oneAPI Math Kernel Library (oneMKL) to dynamically change the number of OpenMP threads.*

Syntax

```
call mkl_set_dynamic( flag )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
flag	INTEGER	*flag* = 0 - Requests disabling dynamic adjustment of the number of threads. *flag*≠ 0 - Requests enabling dynamic adjustment of the number of threads.

Description

This function indicates whether Intel® oneAPI Math Kernel Library (oneMKL) can dynamically change the number of OpenMP threads or should avoid doing this. The setting applies to all Intel® oneAPI Math Kernel Library (oneMKL) functions on all execution threads. This function takes precedence over the`MKL_DYNAMIC` environment variable.

Dynamic adjustment of the number of threads is enabled by default. Specifically, Intel® oneAPI Math Kernel Library (oneMKL) may use fewer threads in parallel regions than the number returned by themkl_get_max_threadsfunction. Disabling dynamic adjustment of the number of threads does not ensure that Intel® oneAPI Math Kernel Library (oneMKL) actually uses the specified number of threads, although the library attempts to use that number.

Tip
If you call Intel® oneAPI Math Kernel Library (oneMKL) from within an OpenMP parallel region and want to create internal parallel regions, either disable dynamic adjustment of the number of threads or set the thread-local number of threads (seemkl_set_num_threads_local for how to do it).

Example

```
use mkl_service
…
call mkl_set_num_threads( 8 )
!$omp parallel
  call my_compute_with_mkl    ! Intel MKL uses 1 thread, being called from OpenMP parallel region
  call mkl_set_dynamic(0)    ! disable adjustment of the number of threads
  call my_compute_with_mkl    ! Intel MKL uses 8 threads
!$omp end parallel
```

mkl_get_max_threads

Gets the number of OpenMP threads targeted for parallelism.*

Syntax

```
nt = mkl_get_max_threads()
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

This function returns the number of OpenMP threads available for Intel® oneAPI Math Kernel Library (oneMKL) to use in internal parallel regions.

Return Values

Name	Type	Description
nt	INTEGER*4	The maximum number of threads for Intel® oneAPI Math Kernel Library (oneMKL) functions to use in internal parallel regions.

Example

```
use mkl_service
…
if (1 == mkl_get_max_threads()) print *, "Intel MKL does not employ threading"
```

See Also

mkl_set_dynamic

mkl_get_dynamic

Using a Fortran Interface Module for Support Functions

mkl_domain_get_max_threads

Gets the number of OpenMP threads targeted for parallelism for a particular function domain.*

Syntax

nt = mkl_domain_get_max_threads(*domain*)

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
domain	INTEGER	The named constant that defines the targeted domain.

Description

Computational functions of the Intel® oneAPI Math Kernel Library (oneMKL) function domain defined by the*domain* parameter use the value returned by this function as a limit of the number of OpenMP threads they should request for parallel computations. The `mkl_domain_get_max_threads` function returns the thread-local number of threads or, if that value is zero or not set, the global number of threads. To determine this number, the function inspects the environment settings and return values of the function calls below in the order they are listed until it finds a non-zero value:

- A call to mkl_set_num_threads_local
- The last of the calls to mkl_set_num_threads or mkl_domain_set_num_threads(…, MKL_DOMAIN_ALL)
- A call to mkl_domain_set_num_threads(…, *domain*)
- The MKL_DOMAIN_NUM_THREADS environment variable with the MKL_DOMAIN_ALL tag
- The MKL_DOMAIN_NUM_THREADS environment variable (with the specific domain tag)
- The MKL_NUM_THREADS environment variable
- A call to omp_set_num_threads
- The OMP_NUM_THREADS environment variable

Actual number of threads used by the Intel® oneAPI Math Kernel Library (oneMKL) computational functions may vary depending on the problem size and on whether dynamic adjustment of the number of threads is enabled (see the description ofmkl_set_dynamic). For a list of supported values of the *domain* argument, see Table "Intel MKL Function Domains".

Return Values

Name	Type	Description
nt	INTEGER*4	The maximum number of threads for Intel® oneAPI Math Kernel Library (oneMKL) functions from a given domain to use in internal parallel regions. If an invalid value of *domain* is supplied, the function returns the number of threads for MKL_DOMAIN_ALL

Example

```
use mkl_service
…
if (1 < mkl_domain_get_max_threads(MKL_DOMAIN_BLAS)) then
print *, "Intel MKL BLAS functions employ threading"
end if
```

mkl_get_dynamic

Determines whether Intel® oneAPI Math Kernel Library (oneMKL) is enabled to dynamically change the number of OpenMP threads.*

Syntax

```
ret = mkl_get_dynamic()
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Description

This function returns the status of dynamic adjustment of the number of OpenMP* threads. To determine this status, the function inspects the return value of the following function call and if it is undefined, inspects the environment setting below:

- A call to mkl_set_dynamic
- The MKL_DYNAMIC environment variable

NOTE
Dynamic adjustment of the number of threads is enabled by default.

The dynamic adjustment works as follows. Suppose that the mkl_get_max_threads function returns the number of threads equal to *N*. If dynamic adjustment is enabled, Intel® oneAPI Math Kernel Library (oneMKL) may request up to*N*threads, depending on the size of the problem. If dynamic adjustment is disabled, Intel® oneAPI Math Kernel Library (oneMKL) requests exactly*N* threads for internal parallel regions (provided it uses a threaded algorithm with at least *N*computations that can be done in parallel). However, the OpenMP* run-time library may be configured to supply fewer threads than Intel® oneAPI Math Kernel Library (oneMKL) requests, depending on the OpenMP* setting of dynamic adjustment.

Return Values

Name	Type	Description
ret	INTEGER*4	0 - Dynamic adjustment of the number of threads is disabled. 1 - Dynamic adjustment of the number of threads is enabled.

Example

```
use mkl_service
integer(4) :: nt
…
nt = mkl_get_max_threads()
if (1 == mkl_get_dynamic()) then
  print '("Intel MKL may use less than "I0" threads for a large problem")', nt
else
  print '("Intel MKL should use "I0" threads for a large problem")', nt
end if
```

mkl_set_num_stripes

Specifies the number of partitions along the leading dimension of the output matrix for parallel ?GEMM *functions.*

Syntax

```
call mkl_set_num_stripes( ns )
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Input Parameters

Name	Type	Description
ns	INTEGER*4	*ns* > 0 - Specifies the number of partitions to use. *ns*= 0 - Instructs Intel® oneAPI Math Kernel Library (oneMKL) to use the default partitioning algorithm.

Name	Type	Description
		ns < 0 - Invalid value; ignored.

Description

This function enables you to specify the number of stripes, or partitions along the leading dimension of the output matrix, for parallel ?GEMMfunctions. If this number is not set (default) or if it is set to zero, Intel® oneAPI Math Kernel Library (oneMKL)?GEMM functions use the default partitioning algorithm. The specified number of partitions only applies to ?GEMM functions.

The number specified is a hint, and Intel® oneAPI Math Kernel Library (oneMKL) may actually use a smaller number.

NOTE
This function takes precedence over the MKL_NUM_STRIPES environment variable.

Example

```
use mkl_service
…
call mkl_set_num_stripes(4)
call dgemm(...)   !Intel MKL uses up to 4 stripes for DGEMM
```

See Also

mkl_get_num_stripes

mkl_get_num_stripes

Gets the number of partitions along the leading dimension of the output matrix for parallel ?GEMM functions.

Syntax

ns = mkl_get_num_stripes( )

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Description

This function returns the number of stripes, that is, partitions along the leading dimension of the output matrix, for parallel ?GEMM functions. The number of partitions only applies to ?GEMM functions.

The number returned is a hint, and Intel® oneAPI Math Kernel Library (oneMKL) may actually use a smaller number.

Return Values

Name	Type	Description
ns	INTEGER*4	The number of stripes for Intel® oneAPI Math Kernel Library (oneMKL)?GEMM functions to use.

Example

```
use mkl_service
…
INTEGER*4 ns = mkl_get_num_stripes()
if (ns .GT. 0) print *, 'Intel MKL uses', ns, 'number of stripes'
```

See Also

mkl_set_num_stripes

Error Handling

Error Handling for Linear Algebra Routines

xerbla

Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.

Syntax

call xerbla(*srname*, *info*)

Include Files

- mkl.fi

Input Parameters

Name	Type	Description
srname	CHARACTER*(*)	The name of the routine that called xerbla
info	INTEGER	The position of the invalid parameter in the parameter list of the calling function or an error code

Description

The xerbla function is an error handler for Intel® oneAPI Math Kernel Library (oneMKL) BLAS, LAPACK, Vector Math, and Vector Statistics functions. These functions call xerbla if an issue is encountered on entry or during the function execution.

xerbla operates as follows:

1. Prints a message that depends on the value of the *info* parameter as explained in the following table.

   NOTE
   A specific message can differ from the listed messages in numeric values and/or function names.

2. Returns to the calling application.

Error Messages Printed by xerbla

Value of *info*	Error Message
1001	`Intel MKL ERROR: Incompatible optional parameters on entry to DGEMM.`
1000 or 1089	`Intel MKL INTERNAL ERROR: Insufficient workspace available in function CGELSD.`
< 0	`Intel MKL INTERNAL ERROR: Condition 1 detected in function DLASD8.`
Other	`The position of the invalid parameter in the parameter list of the calling function.`

Note that `xerbla` is an internal function. You can change or disable printing of an error message by providing your own `xerbla` function. The following examples illustrate usage of `xerbla`.

Example

```
subroutine xerbla (srname, info)
character*(*) srname  !Name of subprogram that called xerbla
integer    info    !Position of the invalid parameter in the parameter list
return    !Return to the calling subprogram end
end
```

See Also

`mkl_set_xerbla`

pxerbla

Error handling routine called by ScaLAPACK routines.

Syntax

`call pxerbla (ictxt, srname, info)`

Include Files

Input Parameters

ictxt

(local)

`INTEGER`

The BLACS context handle, indicating the global context of the operation. The context itself is global.

srname

(global)

`CHARACTER*(*)`

The name of the routine that called `pxerbla`.

info

(global)

`INTEGER`

The position of the invalid parameter in the parameter list of the calling routine.

Description

This routine is an error handler for the *ScaLAPACK* routines. It is called if an input parameter has an invalid value. A message is printed and program execution continues. For ScaLAPACK driver and computational routines, a `RETURN` statement is issued following the call to `pxerbla`.

Control returns to the higher-level calling routine, and you can determine how the program should proceed. However, in the specialized low-level ScaLAPACK routines (auxiliary routines that are Level 2 equivalents of computational routines), the call to `pxerbla()` is immediately followed by a call to `BLACS_ABORT()` to terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a non-zero value of *info* on return from a ScaLAPACK routine. Installers may consider modifying this routine in order to call system-specific exception-handling facilities.

Handling Fatal Errors

A fatal error is a circumstance under which Intel® oneAPI Math Kernel Library (oneMKL) cannot continue the computation. For example, a fatal error occurs when Intel® oneAPI Math Kernel Library (oneMKL) cannot load a dynamic library or confronts an unsupported CPU type. In case of a fatal error, the default Intel® oneAPI Math Kernel Library (oneMKL) behavior is to print an explanatory message to the console and call an internal function that terminates the application with a call to the system`exit()` function. Intel® oneAPI Math Kernel Library (oneMKL) enables you to override this behavior by setting a custom handler of fatal errors. The custom error handler can be configured to throw a C++ exception, set a global variable indicating the failure, or otherwise handle cannot-continue situations. It is not necessary for the custom error handler to call the system`exit()` function. Once execution of the error handler completes, a call to Intel® oneAPI Math Kernel Library (oneMKL) returns to the calling program without performing any computations and leaves no memory allocated by Intel® oneAPI Math Kernel Library (oneMKL) and no thread synchronization pending on return.

To specify a custom fatal error handler, call the `mkl_set_exit_handler` function.

mkl_set_exit_handler

Sets the custom handler of fatal errors.

Syntax

```
external :: myexit
interface = mkl_set_exit_handler( myexit )
```

Fortran Include Files/Modules

None.

Input Parameters

Name	Interface	Description

myexit	`interface`
	`subroutine myexit(iwhy)`
	`integer,value :: iwhy`
	`end subroutine myexit`
	`end interface`
		The error handler to set.

Description

This function sets the custom handler of fatal errors.

The following example shows how to use a custom handler of fatal errors in your application:

```
subroutine myexit(rsn)
integer,value :: rsn
call msgbox("Application is terminating")
```

```
end myexit

program app
external :: myexit
call mkl_set_exit_handler(myexit)
!... compute using Intel MKL...
```

Character Equality Testing

lsame

Tests two characters for equality regardless of the case.

Syntax

val = lsame(*ca*, *cb*)

Include Files

- mkl.fi

Input Parameters

Name	Type	Description
ca, *cb*	CHARACTER*1	The single characters to be compared

Description

This logical function checks whether two characters are equal regardless of the case.

Return Values

Name	Type	Description
val	LOGICAL	Result of the comparison: • .TRUE. if *ca* is the same letter as *cb*, maybe except for the case. • .FALSE. if *ca* and *cb* are different letters for whatever cases.

lsamen

Tests two character strings for equality regardless of the case.

Syntax

val = lsamen(*n*, *ca*, *cb*)

Include Files

- mkl.fi

Input Parameters

Name	Type	Description
n	INTEGER	The number of characters in *ca* and *cb* to be compared.
ca, *cb*	CHARACTER*(*)	Character strings of length at least *n* to be compared. Only the first *n* characters of each string will be accessed.

Description

This logical function tests whether the first *n* letters of one string are the same as the first *n* letters of the other string, regardless of the case.

Return Values

Name	Type	Description
val	LOGICAL	Result of the comparison: • .TRUE. if the first *n* letters in *ca* and *cb* character strings are equal, maybe except for the case, or if the length of character string *ca* or *cb* is less than *n*. • .FALSE. if the first *n* letters in *ca* and *cb* character strings are different for whatever cases.

Timing

second/dsecnd

Returns elapsed time in seconds. Use to estimate real time between two calls to this function.

Syntax

```
val = second()
val = dsecnd()
```

Include Files

- mkl.fi

Description

The second/dsecnd function returns time in seconds to be used to estimate real time between two calls to the function. The difference between these functions is in the precision of the floating-point type of the result: while second returns the single-precision type, dsecnd returns the double-precision type.

Use these functions to measure durations. To do this, call each of these functions twice. For example, to measure performance of a routine, call the appropriate function directly before a call to the routine to be measured, and then after the call of the routine. The difference between the returned values shows real time spent in the routine.

Initializations may take some time when the second/dsecnd function runs for the first time. To eliminate the effect of this extra time on your measurements, make the first call to second/dsecnd in advance.

Do not use second to measure short time intervals because the single-precision format is not capable of holding sufficient timer precision.

Return Values

Name	Type	Description
val	REAL for second DOUBLE PRECISION for dsecnd	Elapsed real time in seconds

mkl_get_cpu_clocks

Returns elapsed CPU clocks.

Syntax

```
call mkl_get_cpu_clocks( clocks )
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Output Parameters

Name	Type	Description
clocks	INTEGER*8	Elapsed CPU clocks

Description

The mkl_get_cpu_clocks function returns the elapsed CPU clocks.

This may be useful when timing short intervals with high resolution. The mkl_get_cpu_clocks function is also applied in pairs like second/dsecnd. Note that out-of-order code execution on IA-32 or Intel® 64 architecture processors may disturb the exact elapsed CPU clocks value a little bit, which may be important while measuring extremely short time intervals.

mkl_get_cpu_frequency

Returns the current CPU frequency value in GHz.

Syntax

```
freq = mkl_get_cpu_frequency()
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Description

The function mkl_get_cpu_frequency returns the current CPU frequency in GHz.

NOTE
The returned value may vary from run to run if power management or Intel® Turbo Boost Technology is enabled.

Return Values

Name	Type	Description
freq	DOUBLE PRECISION	Current CPU frequency value in GHz

mkl_get_max_cpu_frequency

Returns the maximum CPU frequency value in GHz.

Syntax

```
freq = mkl_get_max_cpu_frequency()
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

The function `mkl_get_max_cpu_frequency` returns the maximum CPU frequency in GHz.

Return Values

Name	Type	Description
freq	DOUBLE PRECISION	Maximum CPU frequency value in GHz

mkl_get_clocks_frequency

Returns the frequency value in GHz based on constant-rate Time Stamp Counter.

Syntax

```
freq = mkl_get_clocks_frequency()
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

The function `mkl_get_clocks_frequency` returns the CPU frequency value (in GHz) based on constant-rate Time Stamp Counter (TSC). Use of the constant-rate TSC ensures that each clock tick is constant even if the CPU frequency changes. Therefore, the returned frequency is constant.

NOTE
Obtaining the frequency may take some time when `mkl_get_clocks_frequency` is called for the first time. The same holds for functions `second/dsecnd`, which call `mkl_get_clocks_frequency`.

Return Values

Name	Type	Description
freq	DOUBLE PRECISION	Frequency value in GHz

See Also

`second/dsecnd`
Using a Fortran Interface Module for Support Functions

Memory Management

This section describes the Intel® oneAPI Math Kernel Library (oneMKL) memory functions. See the*Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for more memory usage information.

mkl_free_buffers

Frees unused memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) on the Host.

Syntax

```
call mkl_free_buffers
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

To improve performance of Intel® oneAPI Math Kernel Library (oneMKL) on CPU, the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. Intel® oneAPI Math Kernel Library (oneMKL) also allocates temporary buffers on the host memory to improve performance of GPU kernels. The `mkl_free_buffers` function frees both types of memory.

See the*Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details.

You should call `mkl_free_buffers` after the last call to Intel® oneAPI Math Kernel Library (oneMKL) functions. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel® oneAPI Math Kernel Library (oneMKL) functions.

Usage of mkl_free_buffers with FFT Functions (C Example)

```
DFTI_DESCRIPTOR_HANDLE hand1;
DFTI_DESCRIPTOR_HANDLE hand2;
void mkl_free_buffers(void);
. . . . . .
/* Using Intel MKL FFT */
Status = DftiCreateDescriptor(&hand1, DFTI_SINGLE, DFTI_COMPLEX, dim, m1);
Status = DftiCommitDescriptor(hand1);
Status = DftiComputeForward(hand1, s_array1);
. . . . . .
Status = DftiCreateDescriptor(&hand2, DFTI_SINGLE, DFTI_COMPLEX, dim, m2);
Status = DftiCommitDescriptor(hand2);
. . . . . .
Status = DftiFreeDescriptor(&hand1);
. . . . . .
Status = DftiComputeBackward(hand2, s_array2));
Status = DftiFreeDescriptor(&hand2);
/* Here you finish using Intel MKL FFT */
/* Memory leak will be triggered by any memory control tool */
/* Use mkl_free_buffers() to avoid memory leaking */
mkl_free_buffers();
```

mkl_thread_free_buffers

Frees unused memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator in the current thread.

Syntax

```
call mkl_thread_free_buffers
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

To improve performance of Intel® oneAPI Math Kernel Library (oneMKL), the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The`mkl_thread_free_buffers` function frees unused memory allocated by the Memory Allocator in the current thread only.

You should call `mkl_thread_free_buffers`after the last call to Intel® oneAPI Math Kernel Library (oneMKL) functions in the current thread. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel® oneAPI Math Kernel Library (oneMKL) functions.

See Also

`mkl_free_buffers`

Using a Fortran Interface Module for Support Functions

mkl_disable_fast_mm

Turns off the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator for Intel® oneAPI Math Kernel Library (oneMKL) functions to directly use the system`malloc/free` *functions.*

Syntax

mm = mkl_disable_fast_mm

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Description

The mkl_disable_fast_mmfunction turns the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator off for Intel® oneAPI Math Kernel Library (oneMKL) functions to directly use the systemmalloc/ freefunctions. Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The Memory Allocator is turned on by default for better performance. To turn it off, you can use themkl_disable_fast_mm function or the MKL_DISABLE_FAST_MM environment variable (See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details.) Call mkl_disable_fast_mmbefore calling any Intel® oneAPI Math Kernel Library (oneMKL) functions that require allocation of memory buffers.

> **NOTE**
> Turning the Memory Allocator off negatively impacts performance of some Intel® oneAPI Math Kernel Library (oneMKL) routines, especially for small problem sizes.

Return Values

Name	Type	Description
mm	INTEGER*4	1 - The Memory Allocator is successfully turned off. 0 - Turning the Memory Allocator off failed.

mkl_mem_stat

Reports the status of the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.

Syntax

AllocatedBytes = mkl_mem_stat(*AllocatedBuffers*)

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Output Parameters

Name	Type	Description
AllocatedBuffers	INTEGER*4	The number of buffers allocated by Intel® oneAPI Math Kernel Library (oneMKL).

Description

The function returns the number of buffers allocated by Intel® oneAPI Math Kernel Library (oneMKL) and the amount of memory in these buffers. Intel® oneAPI Math Kernel Library (oneMKL) can allocate the memory buffers internally or in a call tomkl_malloc/mkl_calloc. If no buffers are allocated at the moment, the `mkl_mem_stat` function returns 0. Call `mkl_mem_stat`to check the Intel® oneAPI Math Kernel Library (oneMKL) memory status.

> **NOTE**
> If you free all the memory allocated in calls to `mkl_malloc` or `mkl_calloc` and then call `mkl_free_buffers`, a subsequent call to `mkl_mem_stat` normally returns 0.

Return Values

Name	Type	Description
AllocatedBytes	INTEGER*8	The amount of allocated memory (in bytes).

See Also

Usage Examples for the Memory Functions
Using a Fortran Interface Module for Support Functions

mkl_peak_mem_usage

Reports the peak memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.

Syntax

AllocatedBytes = mkl_peak_mem_usage(*mode*)

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
mode	INTEGER*4	Requested mode of the function's operation. Possible values: • `MKL_PEAK_MEM_ENABLE` - start gathering the peak memory data • `MKL_PEAK_MEM_DISABLE` - stop gathering the peak memory data • `MKL_PEAK_MEM` - return the peak memory • `MKL_PEAK_MEM_RESET` - return the peak memory and reset the counter to start gathering the peak memory data from scratch

Description

The `mkl_peak_mem_usage`function reports the peak memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.

Gathering the peak memory data is turned off by default. If you need to know the peak memory, explicitly turn the data gathering mode on by calling the function with the `MKL_PEAK_MEM_ENABLE` value of the parameter. Use the `MKL_PEAK_MEM` and `MKL_PEAK_MEM_RESET` values only when the data gathering mode is turned on. Otherwise the function returns -1. The data gathering mode leads to performance degradation, so when the mode is turned on, you can turn it off by calling the function with the `MKL_PEAK_MEM_DISABLE` value of the parameter.

NOTE

- If Intel® oneAPI Math Kernel Library (oneMKL) is running in a threaded mode, the`mkl_peak_mem_usage` function may return different amounts of memory from run to run.
- The function reports the peak memory for the entire application, not just for the calling thread.

Return Values

Name	Type	Description
AllocatedBytes	INTEGER*8	The peak memory allocated by the Memory Allocator (in bytes) or -1 in case of errors.

See Also

Usage Examples for the Memory Functions
Using a Fortran Interface Module for Support Functions

mkl_malloc

Allocates an aligned memory buffer.

Syntax

a_ptr = mkl_malloc(*alloc_size*, *alignment*)

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
alloc_size	INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems	Size of the buffer to be allocated.
alignment	INTEGER*4	Alignment of the buffer.

Description

The function allocates an *alloc_size*-byte buffer aligned on the *alignment*-byte boundary.

If *alignment* is not a power of 2, the 64-byte alignment is used.

Return Values

Name	Type	Description
a_ptr	POINTER	Pointer to the allocated buffer if *alloc_size* ≥ 1, NULL if *alloc_size* < 1.

See Also

mkl_free
Usage Examples for the Memory Functions
Using a Fortran Interface Module for Support Functions

mkl_calloc

Allocates and initializes an aligned memory buffer.

Syntax

a_ptr = mkl_calloc(*num*, *size*, *alignment*)

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Input Parameters

Name	Type	Description
num	INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems	The number of elements in the buffer to be allocated.
size	INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems	The size of the element.
alignment	INTEGER*4	Alignment of the buffer.

Description

The function allocates a *num***size*-byte buffer, aligned on the *alignment*-byte boundary, and initializes the buffer with zeros.

If *alignment* is not a power of 2, the 64-byte alignment is used.

Return Values

Name	Type	Description
a_ptr	POINTER	Pointer to the allocated buffer if *size* ≥ 1, NULL if *size* < 1.

See Also

mkl_malloc
mkl_realloc
mkl_free
Usage Examples for the Memory Functions

Using a Fortran Interface Module for Support Functions

mkl_realloc

Changes the size of memory buffer allocated by `mkl_malloc/mkl_calloc`.

Syntax

`a_ptr = mkl_realloc( ptr, size )`

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
ptr	POINTER	Pointer to the memory buffer allocated by the `mkl_malloc` or `mkl_calloc` function or a `NULL` pointer.
size	`INTEGER*4` for 32-bit systems `INTEGER*8` for 64-bit systems	New size of the buffer.

Description

The function changes the size of the memory buffer allocated by the `mkl_malloc` or `mkl_calloc` function to *size* bytes. The first bytes of the returned buffer up to the minimum of the old and new sizes keep the content of the input buffer. The returned memory buffer can have a different location than the input one. If *ptr* is `NULL`, the function works as `mkl_malloc`.

Return Values

Name	Type	Description
a_ptr	POINTER	• Pointer to the re-allocated buffer if re-allocation is successful. • `NULL` if re-allocation is unsuccessful.

See Also

`mkl_malloc`

`mkl_calloc`

`mkl_free`

Usage Examples for the Memory Functions

Using a Fortran Interface Module for Support Functions

mkl_free

Frees the aligned memory buffer allocated by `mkl_malloc/mkl_calloc`.

Syntax

`call mkl_free( a_ptr )`

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
a_ptr	POINTER	Pointer to the buffer to be freed.

Description

The function frees the buffer pointed by *a_ptr* and allocated by the `mkl_malloc()` or `mkl_calloc()` function and does nothing if *a_ptr* is `NULL`.

See Also

`mkl_malloc`

`mkl_calloc`

Usage Examples for the Memory Functions

Using a Fortran Interface Module for Support Functions

mkl_set_memory_limit

On Linux, sets the limit of memory that Intel® oneAPI Math Kernel Library (oneMKL) can allocate for a specified type of memory.

Syntax

```
stat = mkl_set_memory_limit( mem_type, limit )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
mem_type	INTEGER*4	Type of memory to limit. Possible values: `MKL_MEM_MCDRAM` - Multi-Channel Dynamic Random Access Memory (MCDRAM).
limit	INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems.	Memory limit in megabytes.

Description

This function sets the limit for the amount of memory that Intel® oneAPI Math Kernel Library (oneMKL) can allocate for the specified memory type. The limit bounds both internal allocations (inside Intel® oneAPI Math Kernel Library (oneMKL) computation routines) and external allocations (in a call to`mkl_malloc`, `mkl_calloc`, or `mkl_realloc`). By default no limit is set for memory allocation.

Call `mkl_set_memory_limit`at most once, prior to calling any other Intel® oneAPI Math Kernel Library (oneMKL) function in your application except `mkl_set_interface_layer` and `mkl_set_threading_layer`.

NOTE

- Allocation in MCDRAM requires `libmemkind` and `libjemalloc` dynamic libraries which are a part of Intel® Manycore Platform Software Package (Intel® MPSP) for Linux*.
- The `mkl_set_memory_limit` function takes precedence over the `MKL_FAST_MEMORY_LIMIT` environment variable.

Return Values

Type	Description
`INTEGER*4`	Status of the function completion: • 1 - the limit is set • 0 - the limit is not set

See Also

`mkl_malloc`

`mkl_calloc`

`mkl_realloc`

Usage Examples for the Memory Functions

Using a Fortran Interface Module for Support Functions

Usage Examples for the Memory Functions

Usage Example for 1-dimensional Arrays

```
      PROGRAM FOO
      INCLUDE 'mkl.fi'

      DOUBLE PRECISION     A,B,C
      POINTER    (A_PTR,A(1)), (B_PTR,B(1)), (C_PTR,C(1))
      INTEGER    N, I
      REAL*8     ALPHA, BETA
      INTEGER*8  ALLOCATED_BYTES
      INTEGER*8  PEAK_MEMORY
      INTEGER*4  ALLOCATED_BUFFERS

#ifdef _SYSTEM_BITS32
        INTEGER*4 MKL_MALLOC, MKL_CALLOC, MKL_REALLOC
        INTEGER*4 ALLOC_SIZE, NUM, SIZE
#else
        INTEGER*8 MKL_MALLOC, MKL_CALLOC, MKL_REALLOC
        INTEGER*8 ALLOC_SIZE, NUM, SIZE
#endif
```

```
      EXTERNAL   MKL_MALLOC, MKL_FREE, MKL_CALLOC, MKL_REALLOC

      ALPHA = 1.1; BETA = -1.2
      N = 1000
      SIZE = 8
      NUM = N*N
      ALLOC_SIZE = SIZE*NUM
      PEAK_MEMORY = MKL_PEAK_MEM_USAGE(MKL_PEAK_MEM_ENABLE)
      A_PTR = MKL_MALLOC(ALLOC_SIZE,64)
      B_PTR = MKL_MALLOC(ALLOC_SIZE,64)
      C_PTR = MKL_CALLOC(NUM,SIZE,64)
      DO I=1,N*N
        A(I) = I
        B(I) = -I
      END DO

      CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N);

      ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
      PRINT *,'DGEMM uses ',ALLOCATED_BYTES,' bytes in ',
     $  ALLOCATED_BUFFERS,' buffers '

      CALL MKL_FREE_BUFFERS
      CALL MKL_FREE(A_PTR)
      CALL MKL_FREE(B_PTR)
      CALL MKL_FREE(C_PTR)

      ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
      IF (ALLOCATED_BYTES > 0) THEN
       PRINT *,'MKL MEMORY LEAK!'
       PRINT *,'AFTER MKL_FREE_BUFFERS there are ',
     $    ALLOCATED_BYTES,' bytes in ',
     $    ALLOCATED_BUFFERS,' buffers'
      END IF

      PEAK_MEMORY = MKL_PEAK_MEM_USAGE(MKL_PEAK_MEM_RESET)
      PRINT *,'Peak memory allocated by Intel MKL memory allocator ',
     $  PEAK_MEMORY,' bytes. ',
     $  'Start to count new memory peak'

      A_PTR = MKL_MALLOC(ALLOC_SIZE,64)
      A_PTR = MKL_REALLOC(A_PTR,ALLOC_SIZE*SIZE)
      CALL MKL_FREE(A_PTR)
      PEAK_MEMORY = MKL_PEAK_MEM_USAGE(MKL_PEAK_MEM)
      PRINT *,'After reset of peak memory counter',
     $  'Peak memory allocated by Intel MKL memory allocator ',
     $  PEAK_MEMORY,' bytes'

      STOP
      END
```

Usage Example for 2-dimensional Arrays

```
      PROGRAM FOO
      INTEGER    N
      PARAMETER  (N=100)
      DOUBLE PRECISION     A,B,C
      POINTER    (A_PTR,A(N,*)), (B_PTR,B(N,*)), (C_PTR,C(N,*))
      INTEGER    I,J
```

```
      REAL*8     ALPHA, BETA
      INTEGER*8  ALLOCATED_BYTES
      INTEGER*4  ALLOCATED_BUFFERS

#ifdef _SYSTEM_BITS32
        INTEGER*4 MKL_MALLOC
        INTEGER*4 ALLOC_SIZE
#else
        INTEGER*8 MKL_MALLOC
        INTEGER*8 ALLOC_SIZE
#endif

      INTEGER    MKL_MEM_STAT
      EXTERNAL   MKL_MALLOC, MKL_FREE, MKL_MEM_STAT

      ALPHA = 1.1; BETA = -1.2
      ALLOC_SIZE = 8*N*N
      A_PTR = MKL_MALLOC(ALLOC_SIZE,64)
      B_PTR = MKL_MALLOC(ALLOC_SIZE,64)
      C_PTR = MKL_MALLOC(ALLOC_SIZE,64)
      DO I=1,N
        DO J=1,N
           A(I,J) = I
           B(I,J) = -I
           C(I,J) = 0.0
        END DO
      END DO

      CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N);

      ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
      PRINT *,'DGEMM uses ',ALLOCATED_BYTES,' bytes in ',
     $  ALLOCATED_BUFFERS,' buffers '

      CALL MKL_FREE_BUFFERS
      CALL MKL_FREE(A_PTR)
      CALL MKL_FREE(B_PTR)
      CALL MKL_FREE(C_PTR)

      ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
      IF (ALLOCATED_BYTES > 0) THEN
        PRINT *,'MKL MEMORY LEAK!'
        PRINT *,'AFTER MKL_FREE_BUFFERS there are ',
     $    ALLOCATED_BYTES,' bytes in ',
     $    ALLOCATED_BUFFERS,' buffers'
      END IF

      STOP
      END
```

Single Dynamic Library Control

Intel® oneAPI Math Kernel Library (oneMKL) provides the Single Dynamic Library (SDL), which enables setting the interface and threading layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time. See*Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details of SDL and layered model concept. This section describes the functions supporting SDL.

mkl_set_interface_layer

Sets the interface layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time. Use with the Single Dynamic Library.

Syntax

```
interface = mkl_set_interface_layer( required_interface )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
required_interface	INTEGER	Determines the interface layer. Possible values depend on the system architecture. Some of the values are only available on Linux* OS: • Intel® 64 architecture: `MKL_INTERFACE_LP64` for the Intel LP64 interface. `MKL_INTERFACE_ILP64` for the Intel ILP64 interface. `MKL_INTERFACE_LP64+MKL_INTERFACE_GNU` for the GNU* LP64 interface on Linux OS. `MKL_INTERFACE_ILP64+MKL_INTERFACE_GNU` for the GNU ILP64 interface on Linux OS. • IA-32 architecture: `MKL_INTERFACE_LP64` for the Intel interface on Linux OS. `MKL_INTERFACE_LP64+MKL_INTERFACE_GNU` or `MKL_INTERFACE_GNU` for the GNU interface on Linux OS.

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_interface_layer`function sets the specified interface layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time.

Call this function prior to calling any other Intel® oneAPI Math Kernel Library (oneMKL) function in your application except`mkl_set_threading_layer`. You can call `mkl_set_interface_layer` and `mkl_set_threading_layer` in any order.

The `mkl_set_interface_layer` function takes precedence over the `MKL_INTERFACE_LAYER` environment variable.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for the layered model concept and usage details of the SDL.

Return Values

Type	Description
INTEGER	• Current interface layer if it is set in a call to mkl_set_interface_layer or specified by environment variables or defaults. Possible values are specified in Input Parameters. • -1, if the layer was not specified prior to the call and the input parameter is incorrect.

mkl_set_threading_layer

Sets the threading layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time. Use with the Single Dynamic Library (SDL).

Syntax

```
threading = mkl_set_threading_layer( required_threading )
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Input Parameters

Name	Type	Description
required_threading	INTEGER	Determines the threading layer. Possible values: MKL_THREADING_INTEL for Intel threading. MKL_THREADING_SEQUENTIALfor the sequential mode of Intel® oneAPI Math Kernel Library (oneMKL). MKL_THREADING_TBB for threading with the Intel® Threading Building Blocks. MKL_THREADING_PGI for PGI threading on Windows* or Linux* operating system only. Do not use this value with the SDL for Intel® Many Integrated Core (Intel® MIC) Architecture. **NOTE** PGI* support is deprecated and will be removed in the oneMKL 2025.0 release. MKL_THREADING_GNU for GNU threading on Linux* operating system only. Do not use this value with the SDL for Intel MIC Architecture.

Description

If you are using the Single Dynamic Library (SDL), the mkl_set_threading_layerfunction sets the specified threading layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time.

Call this function prior to calling any other Intel® oneAPI Math Kernel Library (oneMKL) function in your application except `mkl_set_interface_layer`.

You can call `mkl_set_threading_layer` and `mkl_set_interface_layer` in any order.

The `mkl_set_threading_layer` function takes precedence over the `MKL_THREADING_LAYER` environment variable.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for the layered model concept and usage details of the SDL.

Return Values

Type	Description
`INTEGER`	• Current threading layer if it is set in a call to `mkl_set_threading_layer` or specified by environment variables or defaults. Possible values are specified in Input Parameters. • -1, if the layer was not specified prior to the call and the input parameter is incorrect.

mkl_set_xerbla

Replaces the error handling routine. Use with the Single Dynamic Library .

Syntax

`old_xerbla_ptr = mkl_set_xerbla( new_xerbla_ptr )`

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
new_xerbla_ptr	`XerblaEntry`	Pointer to the error handling routine to be used.

Description

The `mkl_set_xerbla`function replaces the error handling routine that is called by Intel® oneAPI Math Kernel Library (oneMKL) functions with the routine specified by the parameter.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details about SDL.

Return Values

The function returns the pointer to the replaced error handling routine.

See Also

`xerbla`

Using a Fortran Interface Module for Support Functions

mkl_set_progress

Replaces the progress information routine.

Syntax

```
old_progress_ptr mkl_set_progress( new_progress_ptr )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
new_progress_ptr	ProgressEntry	Pointer to the progress information routine to be used.

Description

The `mkl_set_progress` function replaces the currently used progress information routine with the routine specified by the parameter.

Usually a user-supplied `mkl_progress` function redefines the default `mkl_progress` function automatically. However, you must call `mkl_set_progress` to replace the default `mkl_progress` on Windows* in any of the following cases:

- You are using the Single Dynamic Library (SDL) `mkl_rt.lib`.
- You link dynamically with ScaLAPACK.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details of SDL.

Return Values

The function returns the pointer to the replaced progress information routine.

See Also

`mkl_progress`

Using a Fortran Interface Module for Support Functions

mkl_set_pardiso_pivot

Replaces the routine handling Intel® oneAPI Math Kernel Library (oneMKL) PARDISO pivots with a user-defined routine. Use with the Single Dynamic Library (SDL).

Syntax

```
old_pardiso_pivot_ptr = mkl_set_pardiso_pivot( new_pardiso_pivot_ptr )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
new_pardiso_pivot_ptr	PardisopivotEntry	Pointer to the pivot setting routine to be used.

Description

If you are using the Single Dynamic Library (SDL), the mkl_set_pardiso_pivotfunction replaces the pivot setting routine that is called by Intel® oneAPI Math Kernel Library (oneMKL) functions with the routine specified by the parameter.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for usage details of the SDL.

Return Values

Type	Description
PardisopivotEntry	Pointer to the replaced pivot setting routine.

See Also

mkl_pardiso_pivot

Conditional Numerical Reproducibility Control

The CNR mode of Intel® oneAPI Math Kernel Library (oneMKL) ensures bitwise reproducible results from run to run of Intel® oneAPI Math Kernel Library (oneMKL) functions on a fixed number of threads for a specific Intel instruction set architecture (ISA) under the following conditions:

- Calls to Intel® oneAPI Math Kernel Library (oneMKL) occur in a single executable
- The number of computational threads used by the library does not change in the run

Intel® oneAPI Math Kernel Library (oneMKL) offers both functions and environment variables to support conditional numerical reproducibility. See the*Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for more information on bitwise reproducible results of computations and for details about the environment variables.

The support functions enable you to configure the CNR mode and also provide information on the current and optimal CNR branch on your system. Usage Examples for CNR Support Functions illustrate usage of these functions.

Important
Call the functions that define the behavior of CNR before any of the math library functions that they control.

Intel® oneAPI Math Kernel Library (oneMKL) provides named constants for use as input and output parameters of the functions instead of integer values. SeeNamed Constants for CNR Control for a list of the named constants.

Although you can configure the CNR mode using either the support functions or the environment variables, the functions offer more flexible configuration and control than the environment variables. Settings specified by the functions take precedence over the settings specified by the environment variables.

Use Intel® oneAPI Math Kernel Library (oneMKL) in the CNR mode only in case a need for bitwise reproducible results is critical. Otherwise, run Intel® oneAPI Math Kernel Library (oneMKL) as usual to avoid performance degradation.

While you can supply unaligned input and output data to Intel® oneAPI Math Kernel Library (oneMKL) functions running in the CNR mode, use of aligned data is recommended. Refer toReproducibility Conditions for more details.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

mkl_cbwr_set

Configures the CNR mode of Intel® oneAPI Math Kernel Library (oneMKL).

Syntax

```
status = mkl_cbwr_set( setting )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
setting	INTEGER*4	CNR branch to set. See Named Constants for CNR Control for a list of named constants that specify the settings.

Description

The `mkl_cbwr_set` function configures the CNR mode. In this release, it sets the CNR branch and turns on the CNR mode.

NOTE
Settings specified by the `mkl_cbwr_set` function take precedence over the settings specified by the `MKL_CBWR` environment variable.

Return Values

Name	Type	Description
status	INTEGER*4	The status of the function completion: • `MKL_CBWR_SUCCESS` - the function completed successfully. • `MKL_CBWR_ERR_INVALID_INPUT` - an invalid setting is requested. • `MKL_CBWR_ERR_UNSUPPORTED_BRANCH` - the input value of the branch does not match the instruction set architecture (ISA) of your system. See Named Constants for CNR Control for more details.

Name	Type	Description
		• MKL_CBWR_ERR_MODE_CHANGE_FAILURE - the mkl_cbwr_setfunction requested to change the current CNR branch after a call to some Intel® oneAPI Math Kernel Library (oneMKL) function other than a CNR function.

See Also

Usage Examples for CNR Support Functions
Using a Fortran Interface Module for Support Functions

mkl_cbwr_get

Returns the current CNR settings.

Syntax

```
setting = mkl_cbwr_get( option )
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Input Parameters

Name	Type	Description
option	INTEGER*4	Specifies the CNR settings requested. Named constants define possible values of *option*: • MKL_CBWR_BRANCH - returns the current CNR branch only. • MKL_CBWR_ALL - returns all CNR settings including strict CNR setting.

Description

The mkl_cbwr_get function returns the requested CNR settings. The function returns MKL_CBWR_ERR_INVALID_INPUT if an invalid option is specified.

NOTE
To enable CNR mode, use the mkl_cbwr_set function or environment variables. For more details, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Return Values

Name	Type	Description
setting	INTEGER*4	Requested CNR settings. See Named Constants for CNR Control for a list of named constants that specify the settings.

Name	Type	Description
		If the value of the *option* parameter is not permitted, contains the MKL_CBWR_ERR_INVALID_INPUT error code.

See Also

Usage Examples for CNR Support Functions

mkl_cbwr_set

mkl_cbwr_get_auto_branch

Automatically detects the CNR code branch for your platform.

Syntax

setting = mkl_cbwr_get_auto_branch( )

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Description

The mkl_cbwr_get_auto_branch function uses a run-time CPU check to return a CNR branch that is optimized for the processor where the program is currently running.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Return Values

Name	Type	Description
setting	INTEGER*4	Automatically detected CNR branch. May be any *specific* branch listed in Named Constants for CNR Control.

See Also

Usage Examples for CNR Support Functions

Using a Fortran Interface Module for Support Functions

Named Constants for CNR Control

Use the conditional numerical reproducibility (CNR) functionality in Intel® oneAPI Math Kernel Library (oneMKL) to obtain reproducible results from MKL routines. When enabling CNR, you choose a specific code branch of Intel® oneAPI Math Kernel Library (oneMKL) that corresponds to the instruction set architecture (ISA) that you target. Use these named constants to specify the code branch and other CNR options.

Named Constant	Value	Description
CNR Branches		
MKL_CBWR_OFF	0	Disable CNR mode

Named Constant	Value	Description
MKL_CBWR_BRANCH_OFF	1	CNR mode is disabled
MKL_CBWR_AUTO	2	Choose branch automatically. CNR mode uses the standard ISA-based dispatching model while ensuring fixed cache sizes, deterministic reductions, and static scheduling
MKL_CBWR_COMPATIBLE	3	Intel® Streaming SIMD Extensions 2 (Intel® SSE2) without rcpps/rsqrtps instructions
MKL_CBWR_SSE2	4	Intel SSE2
MKL_CBWR_SSE3	5	DEPRECATED. Intel® Streaming SIMD Extensions 3 (Intel® SSE3). This setting is kept for backward compatibility and is equivalent to MKL_CBWR_SSE2.
MKL_CBWR_SSSE3	6	Supplemental Streaming SIMD Extensions 3 (SSSE3)
MKL_CBWR_SSE4_1	7	Intel® Streaming SIMD Extensions 4-1 (SSE4-1)
MKL_CBWR_SSE4_2	8	Intel® Streaming SIMD Extensions 4-2 (SSE4-2)
MKL_CBWR_AVX	9	Intel® Advanced Vector Extensions (Intel® AVX)
MKL_CBWR_AVX2	10	Intel® Advanced Vector Extensions 2 (Intel® AVX2)
MKL_CBWR_AVX512_MIC	11	DEPRECATED. Intel® Advanced Vector Extensions 512 (Intel® AVX-512) on Intel® Xeon Phi™ processors. This setting is kept for backward compatibility and is equivalent to MKL_CBWR_AVX2.
MKL_CBWR_AVX512	12	Intel AVX-512 on Intel® Xeon® processors
MKL_CBWR_AVX512_MIC_E1	13	DEPRECATED. Intel® Advanced Vector Extensions 512 (Intel® AVX-512) for Intel® Many Integrated Core Architecture (Intel® MIC Architecture) with support of AVX512_4FMAPS and AVX512_4VNNIW instruction groups enabled processors. This setting is kept for backward compatibility and is equivalent to MKL_CBWR_AVX2.
MKL_CBWR_AVX512_E1	14	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support of Vector Neural Network Instructions enabled processors
CNR Flags		
MKL_CBWR_STRICT	65536 or 0x10000	Strict CNR mode enabled. See Reproducibility Conditions for more information.

When specifying the CNR branch with the named constants, be aware of the following:

- Reproducible results are provided under Reproducibility Conditions.
- Settings other than MKL_CBWR_AUTO or MKL_CBWR_COMPATIBLE are available only for Intel processors.
- Intel and Intel compatible CPUs have a few instructions, such as approximation instructions rcpps/rsqrtps, that may return different results. Setting the branch to MKL_CBWR_COMPATIBLEensures that Intel® oneAPI Math Kernel Library (oneMKL) does not use these instructions and forces a single Intel SSE2-only code path to be executed.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

Usage Examples for CNR Support Functions

Reproducibility Conditions

To get reproducible results from run to run, ensure that the number of threads is fixed and constant. Specifically:

- If you are running your program with OpenMP* parallelization on different processors, explicitly specify the number of threads.
- To ensure that your application has deterministic behavior with OpenMP* parallelization and does not adjust the number of threads dynamically at run time, set `MKL_DYNAMIC` and `OMP_DYNAMIC` to FALSE. This is especially needed if you are running your program on different systems.
- If you are running your program with the Intel® Threading Building Blocks parallelization, numerical reproducibility is not guaranteed.

Strict CNR Mode

In strict CNR mode, oneAPI Math Kernel Library provides bitwise reproducible results for a limited set of functions and code branches even when the number of threads changes. These routines and branches support strict CNR mode (64-bit libraries only):

- `?gemm`, `?symm`, `?hemm`, `?trsm`, and their CBLAS equivalents (`cblas_?gemm`, `cblas_?symm`, `cblas_?hemm`, and `cblas_?trsm`.
- Intel® Advanced Vector Extensions 2 (Intel® AVX2) or Intel® Advanced Vector Extensions 512 (Intel® AVX-512).

When using other routines or CNR branches,oneAPI Math Kernel Library operates in standard (non-strict) CNR mode, subject to the restrictions described above. Enabling strict CNR mode can reduce performance.

NOTE

- As usual, you should align your data, even in CNR mode, to obtain the best possible performance. While CNR mode also fully supports unaligned input and output data, the use of it might reduce the performance of some oneAPI Math Kernel Library functions on earlier Intel processors. To ensure proper alignment of arrays, allocate memory for them using `mkl_malloc`/`mkl_calloc`.
- Conditional Numerical Reproducibility does not ensure that bitwise-identical NaN values are generated when the input data contains NaN values.
- If dynamic memory allocation fails on one run but succeeds on another run, you may fail to get reproducible results between these two runs.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

`mkl_malloc`

`mkl_calloc`

Usage Examples for CNR Support Functions

The following examples illustrate usage of support functions for conditional numerical reproducibility.

Setting Automatically Detected CNR Branch

```
    PROGRAM MAIN
    INCLUDE 'mkl.fi'
    INTEGER*4 MY_CBWR_BRANCH
C Find the available MKL_CBWR_BRANCH automatically
    MY_CBWR_BRANCH = MKL_CBWR_GET_AUTO_BRANCH()
C User code without Intel MKL calls
C Piece of the code where CNR of Intel MKL is needed
C The performance of Intel MKL functions might be reduced for CNR mode
    IF (MKL_CBWR_SET (MY_CBWR_BRANCH) .NE. MKL_CBWR_SUCCESS) THEN
        PRINT *, 'Error in setting MKL_CBWR_BRANCH! Aborting…'
        RETURN
    ENDIF
C CNR calls to Intel MKL + any other code
    END
```

Use of the mkl_cbwr_get Function

```
    PROGRAM MAIN
    INCLUDE 'mkl.fi'
    INTEGER*4 MY_CBWR_BRANCH
C Piece of the code where CNR of Intel MKL is analyzed
    MY_CBWR_BRANCH = MKL_CBWR_GET(MKL_CBWR_BRANCH)
    IF (MY_CBWR_BRANCH .EQ. MKL_CBWR_AUTO) THEN
C actions in case of automatic mode
    ELSE IF (MY_CBWR_BRANCH .EQ. MKL_CBWR_SSSE3) THEN
C actions for SSSE3 code
    ELSE
C all other cases
    ENDIF
C User code
    END
```


Miscellaneous

mkl_progress

Provides progress information.

Syntax

stopflag = mkl_progress(*thread_process*, *step*, *stage*)

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
thread_process	INTEGER*4	Indicates the number of thread or process the progress routine is called from: • The thread number for non-cluster components linked with OpenMP threading layer • Zero for non-cluster components linked with sequential threading layer • The process number (MPI rank) for cluster components
step	INTEGER*4	The linear progress indicator that shows the amount of work done. Increases from 0 to the linear size of the problem during the computation.
stage	CHARACTER*(*)	Message indicating the name of the routine or the name of the computation stage the progress routine is called from.

Description

The `mkl_progress` function is intended to track progress of a lengthy computation and/or interrupt the computation. By default this routine does nothing but the user application can redefine it to obtain the computation progress information. You can set it to perform certain operations during the routine computation, for instance, to print a progress indicator. A non-zero return value may be supplied by the redefined function to break the computation.

NOTE
The user-defined `mkl_progress` function must be thread-safe.

Some Intel® oneAPI Math Kernel Library (oneMKL) functions from LAPACK, ScaLAPACK, DSS/PARDISO, and Parallel Direct Sparse Solver for Clusters regularly call the`mkl_progress` function during the computation. Refer to the description of a specific function from those domains to see whether the function supports this feature or not.

If a LAPACK function returns *info*=-1002, the function was interrupted by `mkl_progress`. Because ScaLAPACK does not support interruption of the computation, Intel® oneAPI Math Kernel Library (oneMKL) ignores any value returned by`mkl_progress`.

While a user-supplied `mkl_progress` function usually redefines the default `mkl_progress` function automatically, some configurations require calling the `mkl_set_progress` function to replace the default `mkl_progress` function. Call `mkl_set_progress` to replace the default `mkl_progress` on Windows* in any of the following cases:

- You are using the Single Dynamic Library (SDL) `mkl_rt.lib`.
- You link dynamically with ScaLAPACK.

Warning
The `mkl_progress` function supports OpenMP*/TBB threading and sequential execution for specific routines.

Return Values

Name	Type	Description
stopflag	INTEGER	The stopping flag. A non-zero flag forces the routine to be interrupted. The zero flag is the default return value.

Example

The following example prints the progress information to the standard output device:

```
integer function mkl_progress( thread_process, step, stage )
integer*4 thread_process, step
character*(*) stage
print*,'Thread:',thread_process,',stage:',stage,',step:',step
mkl_progress = 0
return
end
```

mkl_enable_instructions

Enables dispatching for new Intel® architectures or restricts the set of Intel® instruction sets available for dispatching. The `mkl_enable_instructions` function must be called only once, before any other Intel® oneAPI Math Kernel Library (oneMKL) functions.

Syntax

`irc = mkl_enable_instructions(isa)`

Input Parameters

Name	Type	Description
isa	INTEGER*4	The latest Intel® instruction-set architecture (ISA) for Intel® oneAPI Math Kernel Library (oneMKL) to dispatch. `MKL_ENABLE_AVX512` Intel® Advanced Vector Extensions 512 (Intel® AVX-512) `MKL_ENABLE_AVX512_E1` Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for Intel® Deep Learning Boost (Intel® DL Boost). `MKL_ENABLE_AVX512_E2` Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for Intel® Deep Learning Boost (Intel® DL

Name	Type	Description
		Boost), EVEX-encoded AES, and Carry-Less Multiplication Quadword instructions
	`MKL_ENABLE_AVX512_E3`	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for Intel® Deep Learning Boost (Intel® DL Boost) and bfloat16
	`MKL_ENABLE_AVX512_E4`	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for INT8, BF16, FP16 (limited) instructions, and Intel® Advanced Matrix Extensions (Intel® AMX) with INT8 and BF16
	`MKL_ENABLE_AVX512_E5`	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for INT8, BF16, FP16 (limited) instructions, and Intel® Advanced Matrix Extensions (Intel® AMX) with INT8, BF16, and FP16 **NOTE** Not dispatched by default.
	`MKL_ENABLE_AVX2`	Intel® Advanced Vector Extensions 2 (Intel® AVX2)
	`MKL_ENABLE_AVX2_E1`	Intel® Advanced Vector Extensions 2 (Intel® AVX2) with support for Intel® Deep Learning Boost (Intel® DL Boost)
	`MKL_ENABLE_SSE4_2`	Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2)

Description

Intel® oneAPI Math Kernel Library (oneMKL) does run-time processor dispatching to identify appropriate internal code paths to traverse for Intel® oneAPI Math Kernel Library (oneMKL) functions called by the application. The `mkl_enable_instructions` function controls the behavior of the dispatcher to do either of the following:

- Enable dispatching for new Intel architectures.

Intel® oneAPI Math Kernel Library (oneMKL) does not dispatch instruction sets that do not have silicon available at time of the product launch. Call`mkl_enable_instructions` to enable dispatching the code path for such an ISA in a simulator environment or on hardware that supports this ISA.

- Restrict the set of Intel instruction sets available for dispatching.

  Call `mkl_enable_instructions` to restrict dispatching to code paths for earlier ISA. For example, if the hardware supports Intel AVX, a call to `mkl_enable_instructions` with the `MKL_ENABLE_SSE4_2` parameter forces the dispatcher to use the Intel SSE4-2 code path.

If the system does not support the instruction set specified by the *isa* parameter or if the system is based on a non-Intel architecture, `mkl_enable_instructions` does nothing and returns zero.

Settings specified by the `mkl_enable_instructions` function set an upper limit to settings specified by the `mkl_cbwr_set` function.

You can use the `MKL_ENABLE_INSTRUCTIONS` environment variable instead of calling `mkl_enable_instructions` (for more details, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*); however, the settings specified by the function take precedence over the settings specified by the environment variable.

Return Values

Name	Type	Description
irc	INTEGER*4	Function completion status: 1 - Intel® oneAPI Math Kernel Library (oneMKL) dispatches the code path for the specified ISA by default. 0 - The request is rejected. Usually this occurs if `mkl_enable_instructions` was called: • After another Intel® oneAPI Math Kernel Library (oneMKL) function • On a non-Intel architecture • With an incompatible ISA specified

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

mkl_set_env_mode

Sets up the mode that ignores environment settings specific to Intel® oneAPI Math Kernel Library (oneMKL).

Syntax

current_mode = mkl_set_env_mode(*mode*)

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
mode	INTEGER*4	Specifies what mode to set. For details, see Description. Possible values: • 0 - Do nothing. Use this value to query the current environment mode. • 1 - Make Intel® oneAPI Math Kernel Library (oneMKL) ignore environment settings specific to the library.

Description

In the default *environment mode*, Intel® oneAPI Math Kernel Library (oneMKL) can control its behavior using environment variables for threading, memory management, Conditional Numerical Reproducibility, automatic offload, and so on. The`mkl_set_env_mode` function sets up the environment mode that ignores all settings specified by Intel® oneAPI Math Kernel Library (oneMKL) environment variables except`MIC_LD_LIBRARY_PATH` and `MKLROOT`.

Return Values

Name	Type	Description
current_mode	INTEGER*4	Environment mode that was used before the function call: • 0 - Default • 1 - Ignore environment settings specific to Intel® oneAPI Math Kernel Library (oneMKL).

mkl_verbose

Enables or disables Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode.

Syntax

`status = mkl_verbose(`*enable*`)`

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
enable	INTEGER*4	Desired state of the Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode. Indicates whether printing Intel® oneAPI Math Kernel Library (oneMKL) function call information should be turned on or off. Possible values: • 0 – disable the Verbose mode • 1 – enable the Verbose mode (GPU application: enable the Verbose mode without timing)

Name	Type	Description
		• 2 – enable the Verbose mode (GPU application: enable the Verbose mode with synchronous timing)

Description

This function enables or disables the Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode, in which computational functions print call description information. For details of the Verbose mode, see the*Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*, available in the Intel® Software Documentation Library.

NOTE
The setting for the Verbose mode specified by the `mkl_verbose` function takes precedence over the setting specified by the `MKL_VERBOSE` environment variable.

Return Values

Name	Type	Description

status	INTEGER*4	• If the requested operation completed successfully, contains previous state of the verbose mode:
		  • 0 – Verbose mode was disabled
		  • 1 – Verbose mode was enabled (GPU application: Verbose mode was enabled without timing)
		  • 2 – Verbose mode was enabled (GPU application: Verbose mode was enabled with synchronous timing)
		• If the function failed to complete the operation because of an incorrect input parameter, equals –1.

See Also

Intel Software Documentation Library

mkl_verbose_output_file

Write output in Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode to a file.

Syntax

```
mkl_verbose_output_file (filename) character *(*)
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
filename	CHARACTER	Name of file. Specify the complete path of the output file.

Description

This function writes the output in Verbose mode to the file specified in the path.

If the write operation is successful, the function returns 0.

If the file does not exist or cannot be opened, the write operation is unsuccessful. The function returns 1 and defaults to `mkl_verbose` behavior by printing to `stdout`.

NOTE
You can alternatively use MKL_VERBOSE_OUTPUT_FILE environment variable instead of calling the `mkl_verbose_output_file` function. If you want to use the environment variable option, you must set it to the complete path of the output file.

Important The setting for the verbose output file specified by the `mkl_verbose_output_file` function takes precedence over the setting specified by the MKL_VERBOSE_OUTPUT_FILE environment variable.

For more information on the Verbose mode, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*, available in the Intel® Software Documentation Library.

Return Values

Name	Type	Description
status	INTEGER*4	• 0 indicates that the write operation was successful. • 1 indicates that the write operation was unsuccessful.

See Also

Intel Software Documentation Library

mkl_set_mpi

Sets the implementation of the message-passing interface to be used by Intel® oneAPI Math Kernel Library (oneMKL).

Syntax

```
status = mkl_set_mpi(vendor, custom_library_name)
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
vendor	INTEGER*4	Specifies the implementation of the message-passing interface (MPI) to use: Possible values:

Name	Type	Description
		• MKL_BLACS_CUSTOM - a custom MPI library. Requires a prebuilt custom MPI BLACS library. • MKL_BLACS_MSMPI - Microsoft MPI library. • MKL_BLACS_INTELMPI - Intel® MPI library. • MKL_BLACS_MPICH - MPICH MPI library.
custom_library_name	CHARACTER*	The filename (without a directory name) of the custom BLACS dynamic library to use. This library must be located in the directory with your application executable or with Intel® oneAPI Math Kernel Library (oneMKL) dynamic libraries. Can be NULL or an empty string.
vendor		

Description

Call this function to set the MPI implementation to be used by Intel® oneAPI Math Kernel Library (oneMKL) on Windows* OS when dynamic Intel® oneAPI Math Kernel Library (oneMKL) libraries are used. For all other configurations, the function returns an error indicating that you cannot set the MPI implementation. You can specify your own prebuilt dynamic BLACS library for a custom MPI by setting *vendor* to MKL_BLACS_CUSTOM and optionally passing the name of the custom BLACS dynamic library. If the *custom_library_path* parameter is *NULL* or an empty string, Intel® oneAPI Math Kernel Library (oneMKL) uses the default platform-specific library name: mkl_blacs_custom_lp64.dll or mkl_blacs_custom_ilp64.dll, depending on whether the BLACS interface linked against your application is LP64 or ILP64.

Return Values

Name	Type	Description
status	INTEGER*4	The return status: • 0 - The function completed successfully. • -1 - The *vendor* parameter is invalid. • -2 - The *custom_library_name* parameter is invalid. • -3 - The MPI library cannot be set at this point.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

mkl_finalize

Terminates Intel® oneAPI Math Kernel Library (oneMKL) execution environment and frees resources allocated by the library.

Syntax

```
call mkl_finalize
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod

- Module (source): `mkl_service.f90`

Description

This function frees resources allocated by Intel® oneAPI Math Kernel Library (oneMKL). Once this function is called, the application can no longer call Intel® oneAPI Math Kernel Library (oneMKL) functions other than`mkl_finalize`.

In particular, the `mkl_finalize`function enables you to free resources when a third-party shared library is statically linked to Intel® oneAPI Math Kernel Library (oneMKL). To avoid resource leaks that may happen when a shared library is loaded and unloaded multiple times, call`mkl_finalize` each time the library is unloaded. The recommended method to do this depends on the operating system:

- On Linux* or macOS*, place the call into a shared library destructor.
- On Windows*, call `mkl_finalize` from the `DLL_PROCESS_DETACH` handler of `DllMain`.

> **NOTE**
> Intel® oneAPI Math Kernel Library (oneMKL) shared libraries automatically perform finalization when they are unloaded. If an application is statically linked to Intel® oneAPI Math Kernel Library (oneMKL), the operating system frees all resources allocated by Intel® oneAPI Math Kernel Library (oneMKL) during termination of the process associated with the application.

BLACS Routines

Intel® oneAPI Math Kernel Libraryimplements FORTRAN 77 routines from the BLACS (Basic Linear Algebra Communication Subprograms) package. These routines are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The BLACS routines make linear algebra applications both easier to program and more portable. For this purpose, they are used in Intel® oneAPI Math Kernel Library (oneMKL) intended for the Linux* and Windows* OSs as the communication layer of ScaLAPACK and Cluster FFT.

On computers, a linear algebra matrix is represented by a two dimensional array (2D array), and therefore the BLACS operate on 2D arrays. See description of the basic matrix shapes in a special topic.

The BLACS routines implemented in Intel® oneAPI Math Kernel Library (oneMKL) are of four categories:

- Combines
- Point to Point Communication
- Broadcast
- Support.

The Combines take data distributed over processes and combine the data to produce a result. The Point to Point routines are intended for point-to-point communication and Broadcast routines send data possessed by one process to all processes within a scope.

The Support routines perform distinct tasks that can be used for initialization, destruction, information, and miscellaneous tasks.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Matrix Shapes

The BLACS routines recognize the two most common classes of matrices for dense linear algebra. The first of these classes consists of general rectangular matrices, which in machine storage are 2D arrays consisting of *m* rows and *n* columns, with a leading dimension, *lda*, that determines the distance between successive columns in memory.

The *general rectangular* matrices take the following parameters as input when determining what array to operate on:

m	(input) INTEGER. The number of matrix rows to be operated on.
n	(input) INTEGER. The number of matrix columns to be operated on.
a	(input/output) TYPE (depends on routine), array of dimension (*lda*,*n*). A pointer to the beginning of the (sub)array to be sent.
lda	(input) INTEGER. The distance between two elements in matrix row.

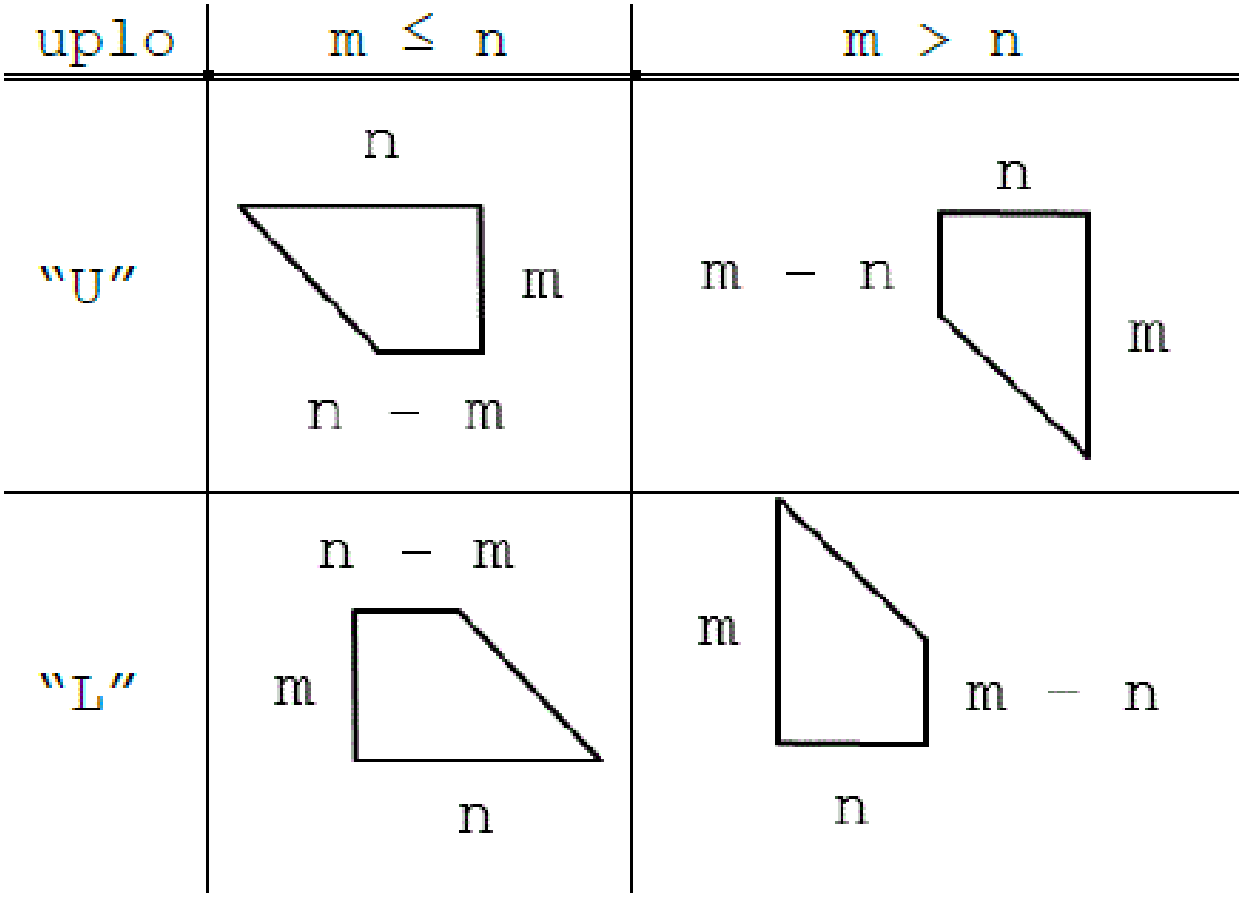
The second class of matrices recognized by the BLACS are *trapezoidal* matrices (triangular matrices are a sub-class of trapezoidal). Trapezoidal arrays are defined by *m*, *n*, and *lda*, as above, but they have two additional parameters as well. These parameters are:

uplo	(input) CHARACTER*1 . Indicates whether the matrix is upper or lower trapezoidal, as discussed below.
diag	(input) CHARACTER*1 . Indicates whether the diagonal of the matrix is unit diagonal (will not be operated on) or otherwise (will be operated on).

The shape of the trapezoidal arrays is determined by these parameters as follows:

__border__top

Trapezoidal Arrays Shapes



The packing of arrays, if required, so that they may be sent efficiently is hidden, allowing the user to concentrate on the logical matrix, rather than on how the data is organized in the system memory.

Repeatability and Coherence

Floating point computations are not exact on almost all modern architectures. This lack of precision is particularly problematic in parallel operations. Since floating point computations are inexact, algorithms are classified according to whether they are *repeatable* and to what degree they guarantee *coherence*.

- Repeatable: a routine is repeatable if it is guaranteed to give the same answer if called multiple times with the same parallel configuration and input.
- Coherent: a routine is coherent if all processes selected to receive the answer get identical results.

> **NOTE**
> Repeatability and coherence do not effect correctness. A routine may be both incoherent and non-repeatable, and still give correct output. But inaccuracies in floating point calculations may cause the routine to return differing values, all of which are equally valid.

Repeatability

Because the precision of floating point arithmetic is limited, it is not truly associative: `(a + b) + c` might not be the same as `a + (b + c)`. The lack of exact arithmetic can cause problems whenever the possibility for reordering of floating point calculations exists. This problem becomes prevalent in parallel computing due to race conditions in message passing. For example, consider a routine which sums numbers stored on different processes. Assume this routine runs on four processes, with the numbers to be added being the process numbers themselves. Therefore, process 0 has the value 0:0, process 1 has the value 1:0, and son on.

One algorithm for the computation of this result is to have all processes send their process numbers to process 0; process 0 adds them up, and sends the result back to all processes. So, process 0 would add a number to 0:0 in the first step. If receiving the process numbers is ordered so that process 0 always receives the message from process 1 first, then 2, and finally 3, this results in a repeatable algorithm, which evaluates the expression `((0:0+ 1:0) + 2:0) + 3:0`.

However, to get the best parallel performance, it is better not to require a particular ordering, and just have process 0 add the first available number to its value and continue to do so until all numbers have been added in. Using this method, a race condition occurs, because the order of the operation is determined by the order in which process 0 receives the messages, which can be effected by any number of things. This implementation is not repeatable, because the answer can vary between invocations, even if the input is the same. For instance, one run might produce the sequence `((0:0+1:0)+2:0)+3:0`, while a subsequent run could produce `((0:0 + 2:0) + 1:0) + 3:0`. Both of these results are correct summations of the given numbers, but because of floating point roundoff, they might be different.

Coherence

A routine produces coherent output if all processes are guaranteed to produce the exact same results. Obviously, almost no algorithm involving communication is coherent if communication can change the values being communicated. Therefore, if the parallel system being studied cannot guarantee that communication between processes preserves values, no routine is guaranteed to produce coherent results.

If communication is assumed to be coherent, there are still various levels of coherent algorithms. Some algorithms guarantee coherence only if floating point operations are done in the exact same order on every node. This is *homogeneous coherence*: the result will be coherent if the parallel machine is homogeneous in its handling of floating point operations.

A stronger assertion of coherence is *heterogeneous coherence*, which does not require all processes to have the same handling of floating point operations.

In general, a routine that is homogeneous coherent performs computations redundantly on all nodes, so that all processes get the same answer only if all processes perform arithmetic in the exact same way, whereas a routine which is heterogeneous coherent is usually constrained to having one process calculate the final result, and broadcast it to all other processes.

Example of Incoherence

An incoherent algorithm is one which does not guarantee that all processes get the same result even on a homogeneous system with coherent communication. The previous example of summing the process numbers demonstrates this kind of behavior. One way to perform such a sum is to have every process broadcast its number to all other processes. Each process then adds these numbers, starting with its own. The calculations performed by each process receives would then be:

- Process 0 : `((0:0+ 1:0) + 2:0) + 3:0`
- Process 1 : `((1:0+ 2:0) + 3:0) + 0:0`
- Process 2 : `((2:0+ 3:0) + 0:0) + 1:0`
- Process 3 : `((3:0+ 0:0) + 1:0) + 0:0`

All of these results are equally valid, and since all the results might be different from each other, this algorithm is incoherent. Notice, however, that this algorithm is repeatable: each process will get the same result if the algorithm is called again on the same data.

Example of Homogeneous Coherence

Another way to perform this summation is for all processes to send their data to all other processes, and to ensure the result is not incoherent, enforce the ordering so that the calculation each node performs is `((0:0+ 1:0) + 2:0) + 3:0`. This answer is the same for all processes only if all processes do the floating point arithmetic in the same way. Otherwise, each process may make different floating point errors during the addition, leading to incoherence of the output. Notice that since there is a specific ordering to the addition, this algorithm is repeatable.

Example of Heterogeneous Coherence

In the final example, all processes send the result to process 0, which adds the numbers and broadcasts the result to the rest of the processes. Since one process does all the computation, it can perform the operations in any order and it will give coherent results as long as communication is itself coherent. If a particular order is not forced on the the addition, the algorithm will not be repeatable. If a particular order is forced, it will be repeatable.

Summary

Repeatability and coherence are separate issues which may occur in parallel computations. These concepts may be summarized as:

- Repeatability: The routine will yield the exact same result if it run multiple times on an identical problem. Each process may get a different result than the others (i.e., repeatability does not imply coherence), but that value will not change if the routine is invoked multiple times.
- Homogeneous coherence: All processes selected to possess the result will receive the exact same answer if:
  - Communication does not change the value of the communicated data.
  - All processes perform floating point arithmetic exactly the same.
- Heterogeneous coherence: All processes will receive the exact same answer if communication does not change the value of the communicated data.

In general, lack of the associative property for floating point calculations may cause both incoherence and non-repeatability. Algorithms that rely on redundant computations are at best homogeneous coherent, and algorithms in which one process broadcasts the result are heterogeneous coherent. Repeatability does not imply coherence, nor does coherence imply repeatability.

Since these issues do not effect the correctness of the answer, they can usually be ignored. However, in very specific situations, these issues may become very important. A stopping criteria should not be based on incoherent results, for instance. Also, a user creating and debugging a parallel program may wish to enforce repeatability so the exact same program sequence occurs on every run.

In the BLACS, coherence and repeatability apply only in the context of the combine operations. As mentioned above, it is possible to have communication which is incoherent (for instance, two machines which store floating point numbers differently may easily produce incoherent communication, since a number stored on machine A may not have a representation on machine B). However, the BLACS cannot control this issue. Communication is assumed to be coherent, which for communication implies that it is also repeatable.

For combine operations, the BLACS allow you to set flags indicating that you would like combines to be repeatable and/or heterogeneous coherent (see `blacs_get` and `blacs_set` for details on setting these flags).

If the BLACS are instructed to guarantee heterogeneous coherency, the BLACS restrict the topologies which can be used so that one process calculates the final result of the combine, and if necessary, broadcasts the answer to all other processes.

If the BLACS are instructed to guarantee repeatability, orderings will be enforced in the topologies which are selected. This may result in loss of performance which can range from negligible to serious depending on the application.

A couple of additional notes are in order. Incoherence and nonrepeatability can arise as a result of floating point errors, as discussed previously. This might lead you to suspect that integer calculations are always repeatable and coherent, since they involve exact arithmetic. This is true if overflow is ignored. With overflow taken into consideration, even integer calculations can display incoherence and non-repeatability. Therefore, if the repeatability or coherence flags are set, the BLACS treats integer combines the same as floating point combines in enforcing repeatability and coherence guards.

By their nature, maximization and minimization should always be repeatable. In the complex precisions, however, the real and imaginary parts must be combined in order to obtain a magnitude value used to do the comparison (this is typically $|r| + |i|$ or $\mathrm{sqr}(r^2 + i^2)$). This allows for the possibility of heterogeneous incoherence. The BLACS therefore restrict which topologies are used for maximization and minimization in the complex routines when the heterogeneous coherence flag is set.

BLACS Combine Operations

This topic describes BLACS routines that combine the data to produce a result.

In a combine operation, each participating process contributes data that is combined with other processes' data to produce a result. This result can be given to a particular process (called the *destination* process), or to all participating processes. If the result is given to only one process, the operation is referred to as a *leave-on-one* combine, and if the result is given to all participating processes the operation is referenced as a *leave-on-all* combine.

At present, three kinds of combines are supported. They are:

- element-wise summation
- element-wise absolute value maximization
- element-wise absolute value minimization

of general rectangular arrays.

Note that a combine operation combines data between processes. By definition, a combine performed across a scope of only one process does not change the input data. This is why the operations (`max`/`min`/`sum`) are specified as *element-wise*. Element-wise indicates that each element of the input array will be combined with the corresponding element from all other processes' arrays to produce the result. Thus, a 4 x 2 array of inputs produces a 4 x 2 answer array.

When the `max`/`min` comparison is being performed, absolute value is used. For example, -5 and 5 are equivalent. However, the returned value is unchanged; that is, it is not the absolute value, but is a signed value instead. Therefore, if you performed a BLACS absolute value maximum combine on the numbers -5, 3, 1, 8 the result would be -8.

The initial symbol `?` in the routine names below masks the data type:

`i`	integer
`s`	single precision real
`d`	double precision real
`c`	single precision complex
`z`	double precision complex.

BLACS Combines

Routine name	Results of operation
gamx2d	Entries of result matrix will have the value of the greatest absolute value found in that position.
gamn2d	Entries of result matrix will have the value of the smallest absolute value found in that position.
gsum2d	Entries of result matrix will have the summation of that position.

?gamx2d

Performs element-wise absolute value maximization.

Syntax

```
call igamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call sgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call dgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call cgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call zgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
```

Input Parameters

icontxt — INTEGER. Integer handle that indicates the context.

scope — CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.

top — CHARACTER*1. Communication pattern to use during the combine operation.

m — INTEGER. The number of matrix rows to be combined.

n — INTEGER. The number of matrix columns to be combined.

a — TYPE array (*lda*, *n*). Matrix to be compared with to produce the maximum.

lda — INTEGER. The leading dimension of the matrix *A*, that is, the distance between two successive elements in a matrix row.

rcflag — INTEGER.

If *rcflag* = -1, the arrays *ra* and *ca* are not referenced and need not exist. Otherwise, *rcflag* indicates the leading dimension of these arrays, and so must be ≥ *m*.

rdest — INTEGER.

The process row coordinate of the process that should receive the result. If *rdest* or *cdest* = -1, all processes within the indicated scope receive the answer.

cdest — INTEGER.

The process column coordinate of the process that should receive the result. If *rdest* or *cdest* = -1, all processes within the indicated scope receive the answer.

Output Parameters

a — TYPE array (*lda*, *n*). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.

ra — INTEGER array (*rcflag*, *n*).

If *rcflag* = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least *rcflag* x *n*) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

ca — INTEGER array (*rcflag*, *n*).

If *rcflag* = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least *rcflag* x *n*) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value maximization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the maximum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

Examples of BLACS Routines Usage

?gamn2d

Performs element-wise absolute value minimization.

Syntax

```
call igamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call sgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call dgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call cgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call zgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
```

Input Parameters

icontxt — INTEGER. Integer handle that indicates the context.

scope — CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.

top CHARACTER*1. Communication pattern to use during the combine operation.

m INTEGER. The number of matrix rows to be combined.

n INTEGER. The number of matrix columns to be combined.

a TYPE array (*lda*, *n*). Matrix to be compared with to produce the minimum.

lda INTEGER. The leading dimension of the matrix *A*, that is, the distance between two successive elements in a matrix row.

rcflag INTEGER.

If *rcflag* = -1, the arrays *ra* and *ca* are not referenced and need not exist. Otherwise, *rcflag* indicates the leading dimension of these arrays, and so must be ≥ *m*.

rdest INTEGER.

The process row coordinate of the process that should receive the result. If *rdest* or *cdest* = -1, all processes within the indicated scope receive the answer.

cdest INTEGER.

The process column coordinate of the process that should receive the result. If *rdest* or *cdest* = -1, all processes within the indicated scope receive the answer.

Output Parameters

a TYPE array (*lda*, *n*). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.

ra INTEGER array (*rcflag*, *n*).

If *rcflag* = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least *rcflag* x *n*) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

ca INTEGER array (*rcflag*, *n*).

If *rcflag* = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least *rcflag* x *n*) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value minimization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the minimum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

Examples of BLACS Routines Usage

?gsum2d

Performs element-wise summation.

Syntax

```
call igsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call sgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call dgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call cgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call zgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
scope	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
top	CHARACTER*1. Communication pattern to use during the combine operation.
m	INTEGER. The number of matrix rows to be combined.
n	INTEGER. The number of matrix columns to be combined.
a	TYPE array (*lda*, *n*). Matrix to be added to produce the sum.
lda	INTEGER. The leading dimension of the matrix *A*, that is, the distance between two successive elements in a matrix row.
rdest	INTEGER. The process row coordinate of the process that should receive the result. If *rdest* or *cdest* = -1, all processes within the indicated scope receive the answer.
cdest	INTEGER. The process column coordinate of the process that should receive the result. If *rdest* or *cdest* = -1, all processes within the indicated scope receive the answer.

Output Parameters

a	TYPE array (*lda*, *n*). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.

Description

This routine performs element-wise summation, that is, each element of matrix *A* is summed with the corresponding element of the other process's matrices. Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

Examples of BLACS Routines Usage

BLACS Point To Point Communication

This topic describes BLACS routines for point to point communication.

Point to point communication requires two complementary operations. The *send* operation produces a message that is then consumed by the *receive* operation. These operations have various resources associated with them. The main such resource is the buffer that holds the data to be sent or serves as the area where the incoming data is to be received. The level of *blocking* indicates what correlation the return from a send/receive operation has with the availability of these resources and with the status of message.

Non-blocking

The return from the *send* or *receive* operations does not imply that the resources may be reused, that the message has been sent/received or that the complementary operation has been called. Return means only that the send/receive has been started, and will be completed at some later date. Polling is required to determine when the operation has finished.

In non-blocking message passing, the concept of *communication/computation overlap* (abbreviated C/C overlap) is important. If a system possesses C/C overlap, independent computation can occur at the same time as communication. That means a nonblocking operation can be posted, and unrelated work can be done while the message is sent/received in parallel. If C/C overlap is not present, after returning from the routine call, computation will be interrupted at some later date when the message is actually sent or received.

Locally-blocking

Return from the *send* or *receive* operations indicates that the resources may be reused. However, since this only depends on local information, it is unknown whether the complementary operation has been called. There are no locally-blocking receives: the send must be completed before the receive buffer is available for re-use.

If a receive has not been posted at the time a locally-blocking send is issued, buffering will be required to avoid losing the message. Buffering can be done on the sending process, the receiving process, or not done at all, losing the message.

Globally-blocking

Return from a globally-blocking procedure indicates that the operation resources may be reused, and that complement of the operation has at least been posted. Since the receive has been posted, there is no buffering required for globally-blocking sends: the message is always sent directly into the user's receive buffer.

Almost all processors support non-blocking communication, as well as some other level of blocking sends. What level of blocking the send possesses varies between platforms. For instance, the Intel® processors support locally-blocking sends, with buffering done on the receiving process. This is a very important distinction, because codes written assuming locally-blocking sends will hang on platforms with globally-blocking sends. Below is a simple example of how this can occur:

```
IAM = MY_PROCESS_ID()
 IF (IAM .EQ. 0) THEN
   SEND TO PROCESS 1
   RECV FROM PROCESS 1
ELSE IF (IAM .EQ. 1) THEN
   SEND TO PROCESS 0
   RECV FROM PROCESS 0
END IF
```

If the send is globally-blocking, process 0 enters the send, and waits for process 1 to start its receive before continuing. In the meantime, process 1 starts to send to 0, and waits for 0 to receive before continuing. Both processes are now waiting on each other, and the program will never continue.

The solution for this case is obvious. One of the processes simply reverses the order of its communication calls and the hang is avoided. However, when the communication is not just between two processes, but rather involves a hierarchy of processes, determining how to avoid this kind of difficulty can become problematic.

For this reason, it was decided the BLACS would support locally-blocking sends. On systems natively supporting globally-blocking sends, non-blocking sends coupled with buffering is used to simulate locally-blocking sends. The BLACS support globally-blocking receives.

In addition, the BLACS specify that point to point messages between two given processes will be strictly ordered. If process 0 sends three messages (label them *A*, *B*, and *C*) to process 1, process 1 must receive *A* before it can receive *B*, and message *C* can be received only after both *A* and *B*. The main reason for this restriction is that it allows for the computation of message identifiers.

Note, however, that messages from different processes are not ordered. If processes 0, . . ., 3 send messages *A*, . . ., *D* to process 4, process 4 may receive these messages in any order that is convenient.

Convention

The convention used in the communication routine names follows the template `?xxyy2d`, where the letter in the `?` position indicates the data type being sent, `xx` is replaced to indicate the shape of the matrix, and the `yy` positions are used to indicate the type of communication to perform:

`i`	integer
`s`	single precision real
`d`	double precision real
`c`	single precision complex
`z`	double precision complex
`ge`	The data to be communicated is stored in a general rectangular matrix.
`tr`	The data to be communicated is stored in a trapezoidal matrix.
`sd`	Send. One process sends to another.
`rv`	Receive. One process receives from another.

BLACS Point To Point Communication

Routine name	Operation performed
gesd2d trsd2d	Take the indicated matrix and send it to the destination process.
gerv2d trrv2d	Receive a message from the process into the matrix.

As a simple example, the pseudo code given above is rewritten below in terms of the BLACS. It is further specifed that the data being exchanged is the double precision vector *X*, which is 5 elements long.

```
CALL GRIDINFO(NPROW, NPCOL, MYPROW, MYPCOL)

IF (MYPROW.EQ.0 .AND. MYPCOL.EQ.0) THEN
   CALL DGESD2D(5, 1, X, 5, 1, 0)
   CALL DGERV2D(5, 1, X, 5, 1, 0)
ELSE IF (MYPROW.EQ.1 .AND. MYPCOL.EQ.0) THEN
   CALL DGESD2D(5, 1, X, 5, 0, 0)
   CALL DGERV2D(5, 1, X, 5, 0, 0)
END IF
```

?gesd2d

Takes a general rectangular matrix and sends it to the destination process.

Syntax

```
call igesd2d( icontxt, m, n, a, lda, rdest, cdest )
call sgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call dgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call cgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call zgesd2d( icontxt, m, n, a, lda, rdest, cdest )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
m, *n*, *a*, *lda*	Describe the matrix to be sent. See Matrix Shapes for details.
rdest	INTEGER. The process row coordinate of the process to send the message to.
cdest	INTEGER. The process column coordinate of the process to send the message to.

Description

This routine takes the indicated general rectangular matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix *A*) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

See Also

Examples of BLACS Routines Usage

?trsd2d

Takes a trapezoidal matrix and sends it to the destination process.

Syntax

```
call itrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call strsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call dtrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ctrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ztrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
uplo, *diag*, *m*, *n*, *a*, *lda*	Describe the matrix to be sent. See Matrix Shapes for details.
rdest	INTEGER. The process row coordinate of the process to send the message to.
cdest	INTEGER. The process column coordinate of the process to send the message to.

Description

This routine takes the indicated trapezoidal matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix *A*) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

?gerv2d

Receives a message from the process into the general rectangular matrix.

Syntax

```
call igerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call sgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call dgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call cgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call zgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
m, *n*, *lda*	Describe the matrix to be sent. See Matrix Shapes for details.
rsrc	INTEGER. The process row coordinate of the source of the message.

csrc	INTEGER. The process column coordinate of the source of the message.

Output Parameters

a	An array of dimension (*lda*,*n*) to receive the incoming message into.

Description

This routine receives a message from process {RSRC, CSRC} into the general rectangular matrix *A*. This routine is globally-blocking, that is, return from the routine indicates that the message has been received into *A*.

See Also

Examples of BLACS Routines Usage

?trrv2d

Receives a message from the process into the trapezoidal matrix.

Syntax

```
call itrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call strrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
uplo, *diag*, *m*, *n*, *lda*	Describe the matrix to be sent. See Matrix Shapes for details.
rsrc	INTEGER. The process row coordinate of the source of the message.
csrc	INTEGER. The process column coordinate of the source of the message.

Output Parameters

a	An array of dimension (*lda*,*n*) to receive the incoming message into.

Description

This routine receives a message from process {RSRC, CSRC} into the trapezoidal matrix *A*. This routine is globally-blocking, that is, return from the routine indicates that the message has been received into *A*.

BLACS Broadcast Routines

This topic describes BLACS broadcast routines.

A broadcast sends data possessed by one process to all processes within a scope. Broadcast, much like point to point communication, has two complementary operations. The process that owns the data to be broadcast issues a *broadcast*/*send*. All processes within the same scope must then issue the complementary *broadcast*/*receive*.

The BLACS define that both broadcast/send and broadcast/receive are *globally-blocking*. Broadcasts/receives cannot be locally-blocking since they must post a receive. Note that receives cannot be locally-blocking. When a given process can leave, a broadcast/receive operation is topology dependent, so, to avoid a hang as topology is varied, the broadcast/receive must be treated as if no process can leave until all processes have called the operation.

Broadcast/sends could be defined to be *locally-blocking*. Since no information is being received, as long as locally-blocking point to point sends are used, the broadcast/send will be locally blocking. However, defining one process within a scope to be locally-blocking while all other processes are globally-blocking adds little to the programmability of the code. On the other hand, leaving the option open to have globally-blocking broadcast/sends may allow for optimization on some platforms.

The fact that broadcasts are defined as globally-blocking has several important implications. The first is that scoped operations (broadcasts or combines) must be strictly ordered, that is, all processes within a scope must agree on the order of calls to separate scoped operations. This constraint falls in line with that already in place for the computation of message IDs, and is present in point to point communication as well.

A less obvious result is that scoped operations with `SCOPE = 'ALL'` must be ordered with respect to any other scoped operation. This means that if there are two broadcasts to be done, one along a column, and one involving the entire process grid, all processes within the process column issuing the column broadcast must agree on which broadcast will be performed first.

The convention used in the communication routine names follows the template `?xxyy2d`, where the letter in the `?` position indicates the data type being sent, `xx` is replaced to indicate the shape of the matrix, and the `yy` positions are used to indicate the type of communication to perform:

`i`	integer
`s`	single precision real
`d`	double precision real
`c`	single precision complex
`z`	double precision complex
`ge`	The data to be communicated is stored in a general rectangular matrix.
`tr`	The data to be communicated is stored in a trapezoidal matrix.
`bs`	Broadcast/send. A process begins the broadcast of data within a scope.
`br`	Broadcast/receive A process receives and participates in the broadcast of data within a scope.

BLACS Broadcast Routines

Routine name	Operation performed
gebs2d trbs2d	Start a broadcast along a scope.
gebr2d trbr2d	Receive and participate in a broadcast along a scope.

Product and Performance Information
Notice revision #20201201

?gebs2d

Starts a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebs2d( icontxt, scope, top, m, n, a, lda )
call sgebs2d( icontxt, scope, top, m, n, a, lda )
call dgebs2d( icontxt, scope, top, m, n, a, lda )
call cgebs2d( icontxt, scope, top, m, n, a, lda )
call zgebs2d( icontxt, scope, top, m, n, a, lda )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
scope	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
top	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
m, *n*, *a*, *lda*	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix *A*.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

See Also

Examples of BLACS Routines Usage

?trbs2d

Starts a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call strbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call dtrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call ctrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call ztrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
scope	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
top	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
uplo, *diag*, *m*, *n*, *a*, *lda*	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix *A*.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

?gebr2d

Receives and participates in a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call sgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call dgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call cgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call zgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
scope	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
top	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
m, *n*, *lda*	Describe the matrix to be sent. See Matrix Shapes for details.
rsrc	INTEGER. The process row coordinate of the process that called broadcast/send.
csrc	INTEGER. The process column coordinate of the process that called broadcast/send.

Output Parameters

a	An array of dimension (*lda*,*n*) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix *A*. Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

See Also

Examples of BLACS Routines Usage

?trbr2d

Receives and participates in a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call strbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
scope	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
top	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
uplo, *diag*, *m*, *n*, *lda*	Describe the matrix to be sent. See Matrix Shapes for details.
rsrc	INTEGER. The process row coordinate of the process that called broadcast/send.
csrc	INTEGER. The process column coordinate of the process that called broadcast/send.

Output Parameters

a	An array of dimension (*lda*,*n*) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix *A*. Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

BLACS Support Routines

The support routines perform distinct tasks that can be used for:

Initialization

Destruction

Information Purposes

Miscellaneous Tasks.

Initialization Routines

This topic describes BLACS routines that deal with grid/context creation, and processing before the grid/context has been defined.

BLACS Initialization Routines

Routine name	Operation performed
blacs_pinfo	Returns the number of processes available for use.
blacs_setup	Allocates virtual machine and spawns processes.
blacs_get	Gets values that BLACS use for internal defaults.
blacs_set	Sets values that BLACS use for internal defaults.
blacs_gridinit	Assigns available processes into BLACS process grid.
blacs_gridmap	Maps available processes into BLACS process grid.

blacs_pinfo

Returns the number of processes available for use.

Syntax

```
call blacs_pinfo( mypnum, nprocs )
```

Output Parameters

mypnum	INTEGER. An integer between 0 and (*nprocs* - 1) that uniquely identifies each process.
nprocs	INTEGER.The number of processes available for BLACS use.

Description

This routine is used when some initial system information is required before the BLACS are set up. On all platforms except PVM, *nprocs* is the actual number of processes available for use, that is, *nprows * npcols <= nprocs*. In PVM, the virtual machine may not have been set up before this call, and therefore no parallel machine exists. In this case, *nprocs* is returned as less than one. If a process has been spawned via the keyboard, it receives *mypnum* of 0, and all other processes get *mypnum* of -1. As a result, the user can distinguish between processes. Only after the virtual machine has been set up via a call to BLACS_SETUP, this routine returns the correct values for *mypnum* and *nprocs*.

See Also

Examples of BLACS Routines Usage

blacs_setup

Allocates virtual machine and spawns processes.

Syntax

```
call blacs_setup( mypnum, nprocs )
```

Input Parameters

nprocs	INTEGER. On the process spawned from the keyboard rather than from pvmspawn, this parameter indicates the number of processes to create when building the virtual machine.

Output Parameters

mypnum	INTEGER. An integer between 0 and (*nprocs* - 1) that uniquely identifies each process.
nprocs	INTEGER. For all processes other than spawned from the keyboard, this parameter means the number of processes available for BLACS use.

Description

This routine only accomplishes meaningful work in the PVM BLACS. On all other platforms, it is functionally equivalent to blacs_pinfo. The BLACS assume a static system, that is, the given number of processes does not change. PVM supplies a dynamic system, allowing processes to be added to the system on the fly.

blacs_setup is used to allocate the virtual machine and spawn off processes. It reads in a file called blacs_setup.dat, in which the first line must be the name of your executable. The second line is optional, but if it exists, it should be a PVM spawn flag. Legal values at this time are 0 (PvmTaskDefault), 4 (PvmTaskDebug), 8 (PvmTaskTrace), and 12 (PvmTaskDebug + PvmTaskTrace). The primary reason for this line is to allow the user to easily turn on and off PVM debugging. Additional lines, if any, specify what machines should be added to the current configuration before spawning *nprocs*-1 processes to the machines in a round robin fashion.

nprocs is input on the process which has no PVM parent (that is, *mypnum*=0), and both parameters are output for all processes. So, on PVM systems, the call to blacs_pinfo informs you that the virtual machine has not been set up, and a call to blacs_setup then sets up the machine and returns the real values for *mypnum* and *nprocs*.

Note that if the file blacs_setup.dat does not exist, the BLACS prompt the user for the executable name, and processes are spawned to the current PVM configuration.

See Also

Examples of BLACS Routines Usage

blacs_get

Gets values that BLACS use for internal defaults.

Syntax

```
call blacs_get( icontxt, what, val )
```

Input Parameters

icontxt — INTEGER. On values of *what* that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.

what — INTEGER. Indicates what BLACS internal(s) should be returned in *val*. Present options are:

- *what* = 0 : Handle indicating default system context.
- *what* = 1 : The BLACS message ID range.
- *what* = 2 : The BLACS debug level the library was compiled with.
- *what* = 10 : Handle indicating the system context used to define the BLACS context whose handle is *icontxt*.
- *what* = 11 : Number of rings multiring broadcast topology is presently using.
- *what* = 12 : Number of branches general tree broadcast topology is presently using.
- *what* = 13 : Number of rings multiring combine topology is presently using.
- *what* = 14 : Number of branches general tree combine topology is presently using.
- *what* = 15 : Whether topologies are forced to be repeatable or not. A non-zero return value indicates that topologies are being forced to be repeatable. See Repeatability and Coherence for more information about repeatability.
- *what* = 16 : Whether topologies are forced to be heterogenous coherent or not. A non-zero return value indicates that topologies are being forced to be heterogenous coherent. See Repeatability and Coherence for more information about coherence.

Output Parameters

val — INTEGER. The value of the BLACS internal.

Description

This routine gets the values that the BLACS are using for internal defaults. Some values are tied to a BLACS context, and some are more general. The most common use is in retrieving a default system context for input into `blacs_gridinit` or `blacs_gridmap`.

Some systems, such as MPI*, supply their own version of context. For those users who mix system code with BLACS code, a BLACS context should be formed in reference to a system context. Thus, the grid creation routines take a system context as input. If you wish to have strictly portable code, you may use `blacs_get` to retrieve a default system context that will include all available processes. This value is not tied to a BLACS context, so the parameter *icontxt* is unused.

`blacs_get` returns information on three quantities that are tied to an individual BLACS context, which is passed in as *icontxt*. The information that may be retrieved is:

- The handle of the system context upon which this BLACS context was defined
- The number of rings for `TOP = 'M'` (multiring broadcast/combine)
- The number of branches for `TOP = 'T'` (general tree broadcast/general tree gather).
- Whether topologies are being forced to be repeatable or heterogenous coherent.

See Also

Examples of BLACS Routines Usage

blacs_set

Sets values that BLACS use for internal defaults.

Syntax

```
call blacs_set( icontxt, what, val )
```

Input Parameters

icontxt

INTEGER. For values of *what* that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.

what

INTEGER. Indicates what BLACS internal(s) should be set. Present values are:

- 1 = Set the BLACS message ID range
- 11 = Number of rings for multiring broadcast topology to use
- 12 = Number of branches for general tree broadcast topology to use
- 13 = Number of rings for multiring combine topology to use
- 14 = Number of branches for general tree combine topology to use
- 15 = Force topologies to be repeatable or not
- 16 = Force topologies to be heterogenous coherent or not

val

INTEGER. Array of dimension (*). Indicates the value(s) the internals should be set to. The specific meanings depend on *what* values, as discussed in Description below.

Description

This routine sets the BLACS internal defaults depending on *what* values:

what = 1

Setting the BLACS message ID range.

If you wish to mix the BLACS with other message-passing packages, restrict the BLACS to a certain message ID range not to be used by the non-BLACS routines. The message ID range must be set before the first call to blacs_gridinit or blacs_gridmap. Subsequent calls will have no effect. Because the message ID range is not tied to a particular context, the parameter *icontxt* is ignored, and *val* is defined as:

VAL (input) INTEGER array of dimension (2)

VAL(1) : The smallest message ID (also called message type or message tag) the BLACS should use.

VAL(2) : The largest message ID (also called message type or message tag) the BLACS should use.

what = 11

Set number of rings for TOP = 'M' (multiring broadcast).This quantity is tied to a context, so *icontxt* is used, and *val* is defined as:

VAL (input) INTEGER array of dimension (1)

VAL(1) : The number of rings for multiring topology to use.

what = 12
Set number of branches for `TOP = 'T'` (general tree broadcast). This quantity is tied to a context, so *icontxt* is used, and *val* is defined as:

`VAL (input) INTEGER` array of dimension (1)

`VAL(1)` : The number of branches for general tree topology to use.

what = 13
Set number of rings for `TOP = 'M'` (multiring combine).This quantity is tied to a context, so *icontxt* is used, and *val* is defined as:

`VAL (input) INTEGER` array of dimension (1)

`VAL(1)` : The number of rings for multiring topology to use.

what = 14
Set number of branches for `TOP = 'T'` (general tree gather). This quantity is tied to a context, so *icontxt* is used, and *val* is defined as:

`VAL (input) INTEGER` array of dimension (1)

`VAL(1)` : The number of branches for general tree topology to use.

what = 15
Force topologies to be repeatable or not (see Repeatability and Coherence for more information about repeatability).

`VAL (input) INTEGER` array of dimension (1)

`VAL(1) = 0` (default)	Topologies are not required to be repeatable.
`VAL(1) ≠ 0`	All used topologies are required to be repeatable, which might degrade performance.

what = 16
Force topologies to be heterogenous coherent or not (see Repeatability and Coherence for more information about coherence).

`VAL (input) INTEGER` array of dimension (1)

`VAL(1) = 0` (default)	Topologies are not required to be heterogenous coherent.
`VAL(1) ≠ 0`	All used topologies are required to be heterogenous coherent, which might degrade performance.

blacs_gridinit

Assigns available processes into BLACS process grid.

Syntax

```
call blacs_gridinit( icontxt, layout, nprow, npcol )
```

Input Parameters

icontxt
`INTEGER`. Integer handle indicating the system context to be used in creating the BLACS context. Call `blacs_get` to obtain a default system context.

layout
`CHARACTER*1`. Indicates how to map processes to BLACS grid. Options are:

- `'R'` : Use row-major natural ordering
- `'C'` : Use column-major natural ordering

- ELSE : Use row-major natural ordering

nprow — INTEGER. Indicates how many process rows the process grid should contain.

npcol — INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

icontxt — INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine blacs_gridmap. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine blacs_setup should be used to create the virtual machine.

This routine creates a simple nprow x npcol process grid. This process grid uses the first nprow * npcol processes, and assigns them to the grid in a row- or column-major natural ordering. If these process-to-grid mappings are unacceptable, call blacs_gridmap.

See Also

Examples of BLACS Routines Usage

blacs_get

blacs_gridmap

blacs_setup

blacs_gridmap

Maps available processes into BLACS process grid.

Syntax

```
call blacs_gridmap( icontxt, usermap, ldumap, nprow, npcol )
```

Input Parameters

icontxt — INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call blacs_get to obtain a default system context.

usermap — INTEGER. Array, dimension (*ldumap*, *npcol*), indicating the process-to-grid mapping.

ldumap	INTEGER. Leading dimension of the 2D array *usermap*. *ldumap* ≥ *nprow*.
nprow	INTEGER. Indicates how many process rows the process grid should contain.
npcol	INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

icontxt	INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridinit`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are actual processors (hardware), and they are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine allows the user to map processes to the process grid in an arbitrary manner. `usermap(i,j)` holds the process number of the process to be placed in `{i, j}` of the process grid. On most distributed systems, this process number is a machine defined number between 0 ... *nprow*-1. For PVM, these node numbers are the PVM TIDS (Task IDs). The `blacs_gridmap` routine is intended for an experienced user. The `blacs_gridinit` routine is much simpler. `blacs_gridinit` simply performs a `gridmap` where the first `nprow * npcol` processes are mapped into the current grid in a row-major natural ordering. If you are an experienced user, `blacs_gridmap` allows you to take advantage of your system's actual layout. That is, you can map nodes that are physically connected to be neighbors in the BLACS grid, etc. The `blacs_gridmap` routine also opens the way for *multigridding*: you can separate your nodes into arbitrary grids, join them together at some later date, and then re-split them into new grids. `blacs_gridmap` also provides the ability to make arbitrary grids or subgrids (for example, a "nearest neighbor" grid), which can greatly facilitate operations among processes that do not fall on a row or column of the main process grid.

See Also

Examples of BLACS Routines Usage

`blacs_get`

`blacs_gridinit`

`blacs_setup`

Destruction Routines

This topic describes BLACS routines that destroy grids, abort processes, and free resources.

BLACS Destruction Routines

Routine name	Operation performed
blacs_freebuff	Frees BLACS buffer.
blacs_gridexit	Frees a BLACS context.
blacs_abort	Aborts all processes.
blacs_exit	Frees all BLACS contexts and releases all allocated memory.

blacs_freebuff

Frees BLACS buffer.

Syntax

```
call blacs_freebuff( icontxt, wait )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the BLACS context.
wait	INTEGER. Parameter indicating whether to wait for non-blocking operations or not. If equals 0, the operations should not be waited for; free only unused buffers. Otherwise, wait in order to free all buffers.

Description

This routine releases the BLACS buffer.

The BLACS have at least one internal buffer that is used for packing messages. The number of internal buffers depends on what platform you are running the BLACS on. On systems where memory is tight, keeping this buffer or buffers may become expensive. Call `freebuff` to release the buffer. However, the next call of a communication routine that requires packing reallocates the buffer.

The *wait* parameter determines whether the BLACS should wait for any non-blocking operations to be completed or not. If *wait* = 0, the BLACS free any buffers that can be freed without waiting. If *wait* is not 0, the BLACS free all internal buffers, even if non-blocking operations must be completed first.

blacs_gridexit

Frees a BLACS context.

Syntax

```
call blacs_gridexit( icontxt )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the BLACS context to be freed.

Description

This routine frees a BLACS context.

Release the resources when contexts are no longer needed. After freeing a context, the context no longer exists, and its handle may be re-used if new contexts are defined.

blacs_abort

Aborts all processes.

Syntax

```
call blacs_abort( icontxt, errornum )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the BLACS context to be aborted.
errornum	INTEGER. User-defined integer error number.

Description

This routine aborts all the BLACS processes, not only those confined to a particular context.

Use `blacs_abort` to abort all the processes in case of a serious error. Note that both parameters are input, but the routine uses them only in printing out the error message. The context handle passed in is not required to be a valid context handle.

blacs_exit

Frees all BLACS contexts and releases all allocated memory.

Syntax

```
call blacs_exit( continue )
```

Input Parameters

continue	INTEGER. Flag indicating whether message passing continues after the BLACS are done. If *continue* is non-zero, the user is assumed to continue using the machine after completing the BLACS. Otherwise, no message passing is assumed after calling this routine.

Description

This routine frees all BLACS contexts and releases all allocated memory.

This routine should be called when a process has finished all use of the BLACS. The *continue* parameter indicates whether the user will be using the underlying communication platform after the BLACS are finished. This information is most important for the PVM BLACS. If *continue* is set to 0, then `pvm_exit` is called; otherwise, it is not called. Setting *continue* not equal to 0 indicates that explicit PVM `send/recvs` will be called after the BLACS routines are used. Make sure your code calls `pvm_exit`. PVM users should either call `blacs_exit` or explicitly call `pvm_exit` to avoid PVM problems.

See Also

Examples of BLACS Routines Usage

Informational Routines

This topic describes BLACS routines that return information involving the process grid.

BLACS Informational Routines

Routine name	Operation performed
blacs_gridinfo	Returns information on the current grid.
blacs_pnum	Returns the system process number of the process in the process grid.
blacs_pcoord	Returns the row and column coordinates in the process grid.

blacs_gridinfo

Returns information on the current grid.

Syntax

```
call blacs_gridinfo( icontxt, nprow, npcol, myprow, mypcol )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.

Output Parameters

nprow	INTEGER. Number of process rows in the current process grid.
npcol	INTEGER. Number of process columns in the current process grid.
myprow	INTEGER. Row coordinate of the calling process in the process grid.
mypcol	INTEGER. Column coordinate of the calling process in the process grid.

Description

This routine returns information on the current grid. If the context handle does not point at a valid context, all quantities are returned as -1.

See Also

Examples of BLACS Routines Usage

blacs_pnum

Returns the system process number of the process in the process grid.

Syntax

```
call blacs_pnum( icontxt, prow, pcol )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
prow	INTEGER. Row coordinate of the process the system process number of which is to be determined.
pcol	INTEGER. Column coordinate of the process the system process number of which is to be determined.

Description

This function returns the system process number of the process at {PROW, PCOL} in the process grid.

See Also

Examples of BLACS Routines Usage

blacs_pcoord

Returns the row and column coordinates in the process grid.

Syntax

```
call blacs_pcoord( icontxt, pnum, prow, pcol )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
pnum	INTEGER. Process number the coordinates of which are to be determined. This parameter stand for the process number of the underlying machine, that is, it is a `tid` for PVM.

Output Parameters

prow	INTEGER. Row coordinates of the *pnum* process in the BLACS grid.
pcol	INTEGER. Column coordinates of the *pnum* process in the BLACS grid.

Description

Given the system process number, this function returns the row and column coordinates in the BLACS process grid.

See Also

Examples of BLACS Routines Usage

Miscellaneous Routines

This topic describes `blacs_barrier` routine.

BLACS Informational Routines

Routine name	Operation performed
blacs_barrier	Holds up execution of all processes within the indicated scope until they have all called the routine.

blacs_barrier

Holds up execution of all processes within the indicated scope.

Syntax

```
call blacs_barrier( icontxt, scope )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.

scope	CHARACTER*1. Parameter that indicates whether a process row (*scope*='R'), column ('C'), or entire grid ('A') will participate in the barrier.

Description

This routine holds up execution of all processes within the indicated scope until they have all called the routine.

Examples of BLACS Routines Usage

Example. BLACS Usage. Hello World

The following routine takes the available processes, forms them into a process grid, and then has each process check in with the process at {0,0} in the process grid.

```
      PROGRAM HELLO
*     -- BLACS example code --
*     Written by Clint Whaley 7/26/94
*     Performs a simple check-in type hello world
*     ..
*     .. External Functions ..
      INTEGER BLACS_PNUM
      EXTERNAL BLACS_PNUM
*     ..
*     .. Variable Declaration ..
      INTEGER CONTXT, IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL
      INTEGER ICALLER, I, J, HISROW, HISCOL
*
*     Determine my process number and the number of processes in
*     machine
*
      CALL BLACS_PINFO(IAM, NPROCS)
*
*     If in PVM, create virtual machine if it doesn't exist
*
      IF (NPROCS .LT. 1) THEN
         IF (IAM .EQ. 0) THEN
            WRITE(*, 1000)
            READ(*, 2000) NPROCS
         END IF
         CALL BLACS_SETUP(IAM, NPROCS)
      END IF
*
*     Set up process grid that is as close to square as possible
*
      NPROW = INT( SQRT( REAL(NPROCS) ) )
      NPCOL = NPROCS / NPROW
*
*     Get default system context, and define grid

*
      CALL BLACS_GET(0, 0, CONTXT)
      CALL BLACS_GRIDINIT(CONTXT, 'Row', NPROW, NPCOL)
      CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYPROW, MYPCOL)
*
*     If I'm not in grid, go to end of program
*
```

```
      IF ( (MYPROW.GE.NPROW) .OR. (MYPCOL.GE.NPCOL) ) GOTO 30

*
*     Get my process ID from my grid coordinates
*
      ICALLER = BLACS_PNUM(CONTXT, MYPROW, MYPCOL)
*
*     If I am process {0,0}, receive check-in messages from
*     all nodes
*
      IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN

         WRITE(*,*) ' '

         DO 20 I = 0, NPROW-1
            DO 10 J = 0, NPCOL-1

               IF ( (I.NE.0) .OR. (J.NE.0) ) THEN
                  CALL IGERV2D(CONTXT, 1, 1, ICALLER, 1, I, J)
               END IF
*
*              Make sure ICALLER is where we think in process grid

*
              CALL BLACS_PCOORD(CONTXT, ICALLER, HISROW, HISCOL)
              IF ( (HISROW.NE.I) .OR. (HISCOL.NE.J) ) THEN
                 WRITE(*,*) 'Grid error!  Halting . . .'

                 STOP
              END IF
              WRITE(*, 3000) I, J, ICALLER



10          CONTINUE
20       CONTINUE
         WRITE(*,*) ' '
         WRITE(*,*) 'All processes checked in.  Run finished.'
*
*     All processes but {0,0} send process ID as a check-in


*
      ELSE

         CALL IGESD2D(CONTXT, 1, 1, ICALLER, 1, 0, 0)
      END IF


30    CONTINUE



      CALL BLACS_EXIT(0)

1000  FORMAT('How many processes in machine?')
2000  FORMAT(I)
3000  FORMAT('Process {',i2,',',i2,'} (node number =',I,
```

```
     $       ') has checked in.')

      STOP
      END
```

Example. BLACS Usage. PROCMAP

This routine maps processes to a grid using `blacs_gridmap`.

```
      SUBROUTINE PROCMAP(CONTEXT, MAPPING, BEGPROC, NPROW, NPCOL, IMAP)
*
*     -- BLACS example code --

*     Written by Clint Whaley 7/26/94
*     ..
*     .. Scalar Arguments ..
      INTEGER CONTEXT, MAPPING, BEGPROC, NPROW, NPCOL

*     ..
*     .. Array Arguments ..
      INTEGER IMAP(NPROW, *)
*     ..
*
*  Purpose
*  =======
*  PROCMAP maps NPROW*NPCOL processes starting from process BEGPROC to
*  the grid in a variety of ways depending on the parameter MAPPING.

*
*  Arguments

*  =========
*
*  CONTEXT      (output) INTEGER
*               This integer is used by the BLACS to indicate a context.
*               A context is a universe where messages exist and do not
*               interact with other context's messages.  The context
*               includes the definition of a grid, and each process's
*               coordinates in it.
*
*  MAPPING      (input) INTEGER
*               Way to map processes to grid.  Choices are:
*               1 : row-major natural ordering
*               2 : column-major natural ordering
*
*  BEGPROC      (input) INTEGER
*               The process number (between 0 and NPROCS-1) to use as

*               {0,0}. From this process, processes will be assigned
*               to the grid as indicated by MAPPING.
*
*  NPROW        (input) INTEGER
*               The number of process rows the created grid

*               should have.
*
*  NPCOL        (input) INTEGER
*               The number of process columns the created grid
```

```
*               should have.
*
*  IMAP         (workspace) INTEGER array of dimension (NPROW, NPCOL)
*               Workspace, where the array which maps the

*               processes to the grid will be stored for the
*               call to GRIDMAP.
*
*     ===========================================================
*
*     ..
*     .. External Functions ..
      INTEGER  BLACS_PNUM

      EXTERNAL BLACS_PNUM

*     ..
*     .. External Subroutines ..
      EXTERNAL BLACS_PINFO, BLACS_GRIDINIT, BLACS_GRIDMAP
*     ..
*     .. Local Scalars ..
      INTEGER TMPCONTXT, NPROCS, I, J, K

*     ..
*     .. Executable Statements ..
*
*     See how many processes there are in the system
*
      CALL BLACS_PINFO( I, NPROCS )

      IF (NPROCS-BEGPROC .LT. NPROW*NPCOL) THEN
         WRITE(*,*) 'Not enough processes for grid'
         STOP
      END IF
*
*     Temporarily map all processes into 1 x NPROCS grid



*
      CALL BLACS_GET( 0, 0, TMPCONTXT )
      CALL BLACS_GRIDINIT( TMPCONTXT, 'Row', 1, NPROCS )
      K = BEGPROC



*
*     If we want a row-major natural ordering



*
      IF (MAPPING .EQ. 1) THEN

         DO I = 1, NPROW
            DO J = 1, NPCOL
               IMAP(I, J) = BLACS_PNUM(TMPCONTXT, 0, K)
               K = K + 1W
            END DO
         END DO
*
```

```
*     If we want a column-major natural ordering


*
      ELSE IF (MAPPING .EQ. 2) THEN

         DO J = 1, NPCOL
            DO I = 1, NPROW
               IMAP(I, J) = BLACS_PNUM(TMPCONTXT, 0, K)

               K = K + 1

            END DO
         END DO
      ELSE

         WRITE(*,*) 'Unknown mapping.'
         STOP
      END IF

*
*     Free temporary context
*
      CALL BLACS_GRIDEXIT(TMPCONTXT)
*
*     Apply the new mapping to form desired context
*
      CALL BLACS_GET( 0, 0, CONTEXT )
      CALL BLACS_GRIDMAP( CONTEXT, IMAP, NPROW, NPROW, NPCOL )




      RETURN
      END
```

Example. BLACS Usage. PARALLEL DOT PRODUCT

This routine does a bone-headed parallel double precision dot product of two vectors. Arguments are input on process {0,0}, and output everywhere else.

```
      DOUBLE PRECISION FUNCTION PDDOT( CONTEXT, N, X, Y )
*
*     -- BLACS example code --

*     Written by Clint Whaley 7/26/94
*     ..
*     .. Scalar Arguments ..
      INTEGER CONTEXT, N

*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION X(*), Y(*)
*     ..
*
*  Purpose
*  =======
*  PDDOT is a restricted parallel version of the BLAS routine
*  DDOT.  It assumes that the increment on both vectors is one,
```

```
*  and that process {0,0} starts out owning the vectors and

*  has N.  It returns the dot product of the two N-length vectors
*  X and Y, that is, PDDOT = X' Y.
*
*  Arguments

*  =========
*
*  CONTEXT      (input) INTEGER
*               This integer is used by the BLACS to indicate a context.
*               A context is a universe where messages exist and do not
*               interact with other context's messages.  The context
*               includes the definition of a grid, and each process's
*               coordinates in it.
*
*  N            (input/output) INTEGER
*               The length of the vectors X and Y. Input
*               for {0,0}, output for everyone else.
*
*  X            (input/output) DOUBLE PRECISION array of dimension (N)
*               The vector X of PDDOT = X' Y. Input for {0,0},
*               output for everyone else.
*
*  Y            (input/output) DOUBLE PRECISION array of dimension (N)
*               The vector Y of PDDOT = X' Y. Input for {0,0},
*               output for everyone else.
*
*     =================================================================
*
*     ..
*     .. External Functions ..
      DOUBLE PRECISION DDOT

      EXTERNAL DDOT

*     ..
*     .. External Subroutines ..
      EXTERNAL BLACS_GRIDINFO, DGEBS2D, DGEBR2D, DGSUM2D
*     ..
*     .. Local Scalars ..
      INTEGER IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL, I, LN


      DOUBLE PRECISION LDDOT



*     ..
*     .. Executable Statements ..
*
*     Find out what grid has been set up, and pretend it is 1-D
*
      CALL BLACS_GRIDINFO( CONTXT, NPROW, NPCOL, MYPROW, MYPCOL )

      IAM = MYPROW*NPCOL + MYPCOL
      NPROCS = NPROW * NPCOL
*
*     Temporarily map all processes into 1 x NPROCS grid
```

```
*
      CALL BLACS_GET( 0, 0, TMPCONTXT )
      CALL BLACS_GRIDINIT( TMPCONTXT, 'Row', 1, NPROCS )
      K = BEGPROC


*
*     Do bone-headed thing, and just send entire X and Y to


*     everyone


*
      IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN

         CALL IGEBS2D(CONTXT, 'All', 'i-ring', 1, 1, N, 1 )

         CALL DGEBS2D(CONTXT, 'All', 'i-ring', N, 1, X, N )
         CALL DGEBS2D(CONTXT, 'All', 'i-ring', N, 1, Y, N )
      ELSE
         CALL IGEBR2D(CONTXT, 'All', 'i-ring', 1, 1, N, 1, 0, 0 )
         CALL DGEBR2D(CONTXT, 'All', 'i-ring', N, 1, X, N, 0, 0 )
         CALL DGEBR2D(CONTXT, 'All', 'i-ring', N, 1, Y, N, 0, 0 )
      ENDIF
*
*     Find out the number of local rows to multiply (LN), and


*     where in vectors to start (I)


*
      LN = N / NPROCS

      I = 1 + IAM * LN
*
*     Last process does any extra rows
*
      IF (IAM .EQ. NPROCS-1) LN = LN + MOD(N, NPROCS)
*
*     Figure dot product of my piece of X and Y

*
      LDDOT = DDOT( LN, X(I), 1, Y(I), 1 )
*
*     Add local dot products to get global dot product;


*     give all procs the answer


*
      CALL DGSUM2D( CONTXT, 'All', '1-tree', 1, 1, LDDOT, 1, -1, 0 )
```

```
      PDDOT = LDDOT

      RETURN

      END
```

Example. BLACS Usage. PARALLEL MATRIX INFINITY NORM

This routine does a parallel infinity norm on a distributed double precision matrix. Unlike the PDDOT example, this routine assumes the matrix has already been distributed.

```
      DOUBLE PRECISION FUNCTION PDINFNRM(CONTXT, LM, LN, A, LDA, WORK)
*
*     -- BLACS example code --

*     Written by Clint Whaley.
*     ..
*     .. Scalar Arguments ..
      INTEGER CONTEXT, LM, LN, LDA


*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION A(LDA, *), WORK(*)
*     ..
*
*  Purpose
*  =======
*  Compute the infinity norm of a distributed matrix, where
*  the matrix is spread across a 2D process grid.  The result is
*  left on all processes.

*
*  Arguments

*  =========
*
*  CONTEXT      (input) INTEGER
*               This integer is used by the BLACS to indicate a context.
*               A context is a universe where messages exist and do not
*               interact with other context's messages.  The context
*               includes the definition of a grid, and each process's
*               coordinates in it.
*
*  LM           (input) INTEGER
*               Number of rows of the global matrix owned by this
*               process.
*
*  LN           (input) INTEGER
*               Number of columns of the global matrix owned by this
*               process.
*
*  A            (input) DOUBLE PRECISION, dimension (LDA,N)
*               The matrix whose norm you wish to compute.

*
*  LDA          (input) INTEGER
*               Leading Dimension of A.
```

```
*
*  WORK         (temporary) DOUBLE PRECISION array, dimension (LM)
*               Temporary work space used for summing rows.


*
*     .. External Subroutines ..
      EXTERNAL BLACS_GRIDINFO, DGEBS2D, DGEBR2D, DGSUM2D, DGAMX2D

*     ..
*     .. External Functions ..
      INTEGER IDAMAX
      DOUBLE PRECISION DASUM
*
*     .. Local Scalars ..
      INTEGER NPROW, NPCOL, MYROW, MYCOL,  I, J



      DOUBLE PRECISION MAX


*
*     .. Executable Statements ..
*
*     Get process grid information
*
      CALL BLACS_GRIDINFO( CONTXT, NPROW, NPCOL, MYPROW, MYPCOL )

*
*     Add all local rows together


*
      DO 20 I = 1, LM

         WORK(I) = DASUM(LN, A(I,1), LDA)
20    CONTINUE


*
*     Find sum of global matrix rows and store on column 0 of


*     process grid


*
      CALL DGSUM2D(CONTXT, 'Row', '1-tree', LM, 1, WORK, LM, MYROW, 0)


*
*     Find maximum sum of rows for supnorm


*
      IF (MYCOL .EQ. 0) THEN
```

```
         MAX = WORK(IDAMAX(LM,WORK,1))

         IF (LM .LT. 1) MAX = 0.0D0


         CALL DGAMX2D(CONTXT, 'Col', 'h', 1, 1, MAX, 1, I, I, -1, -1, 0)
      END IF

*
*     Process column 0 has answer; send answer to all nodes

*
      IF (MYCOL .EQ. 0) THEN

         CALL DGEBS2D(CONTXT, 'Row', ' ', 1, 1, MAX, 1)

      ELSE



         CALL DGEBR2D(CONTXT, 'Row', ' ', 1, 1, MAX, 1, 0, 0)

      END IF

*
      PDINFNRM = MAX

*
      RETURN
*
*     End of PDINFNRM


*
      END
```

Data Fitting Functions

Data Fitting functions in Intel® oneAPI Math Kernel Library (oneMKL) provide spline-based interpolation capabilities that you can use to approximate functions, function derivatives or integrals, and perform cell search operations.

The Data Fitting component is task based. The task is a data structure or descriptor that holds the parameters related to a specific Data Fitting operation. You can modify the task parameters using the task editing functionality of the library.

For definition of the implemented operations, see Mathematical Conventions.

Data Fitting routines use the following workflow to process a task:

1. Create a task or multiple tasks.
2. Modify the task parameters.
3. Perform a Data Fitting computation.
4. Destroy the task or tasks.

All Data Fitting functions fall into the following categories:

Task Creation and Initialization Routines - routines that create a new Data Fitting task descriptor and initialize the most common parameters, such as partition of the interpolation interval, values of the vector-valued function, and the parameters describing their structure.

Task Configuration Routines - routines that set, modify, or query parameters in an existing Data Fitting task.

Computational Routines - routines that perform Data Fitting computations, such as construction of a spline, interpolation, computation of derivatives and integrals, and search.

Task Destructors - routines that delete Data Fitting task descriptors and deallocate resources.

You can access the Data Fitting routines through the Fortran and C89/C99 language interfaces. You can also use the C89 interface with more recent versions of C/C++, or the Fortran 90 interface with programs written in Fortran 95.

The `${MKL}/include`directory of the Intel® oneAPI Math Kernel Library (oneMKL) contains the following Data Fitting header files:

- `mkl_df.f90`

You can find examples that demonstrate usage of Data Fitting routines in the `${MKL}/examples/datafittingf` directory.

Data Fitting Function Naming Conventions

The interfaces of the Data Fitting functions are in lowercase, while the names of the types and constants are in uppercase.

The names of all routines have the following structure:

`df[`*`datatype`*`]<`*`base_name`*`>`

where

- `df`is a prefix indicating that the routine belongs to the Data Fitting component of Intel® oneAPI Math Kernel Library (oneMKL).
- `[`*`datatype`*`]` field specifies the type of the input and/or output data and can be `s` (for the single precision real type), `d` (for the double precision real type), or `i` (for the integer type). This field is omitted in the names of the routines that are not data type dependent.
- `<`*`base_name`*`>` field specifies the functionality the routine performs. For example, this field can be `newtask1d`, `interpolate1d`, or `deletetask`

Data Fitting Function Data Types

The Data Fitting component provides routines for processing single and double precision real data types. The results of cell search operations are returned as a generic integer data type.

All Data Fitting routines use the following data type:

Type	Data Object
`TYPE(DF_TASK)`	Pointer to a task

NOTE
The actual size of the generic integer type is platform-dependent. Before compiling your application, you need to set an appropriate byte size for integers. For details, see section *Using the ILP64 Interface vs. LP64 Interface* of the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Mathematical Conventions for Data Fitting Functions

This section explains the notation used for Data Fitting function descriptions. Spline notations are based on the terminology and definitions of [deBoor2001]. The Subbotin quadratic spline definition follows the conventions of [StechSub76]. The quasi-uniform partition definition is based on [Schumaker2007].

Mathematical Notation in the Data Fitting Component

Concept	Mathematical Notation
Partition of interpolation interval $[a, b]$, where • x_i denotes breakpoints. • $[x_i, x_{i+1})$ denotes a sub-interval (cell) of size $\Delta_i=x_{i+1}-x_i$.	$\{x_i\}_{i=1,...,n}$, where $a = x_1 < x_2 < ... < x_n = b$
Quasi-uniform partition of interpolation interval $[a, b]$	Partition $\{x_i\}_{i=1,...,n}$ which meets the constraint with a constant C defined as $1 \le M/m \le C$, where • $M = \max_{i=1,...,n-1} (\Delta_i)$ • $m = \min_{i=1,...,n-1} (\Delta_i)$ • $\Delta_i = x_{i+1} - x_i$
Vector-valued function of dimension p being fit	$f(x) = (f_1(x),..., f_p(x))$
Piecewise polynomial (PP) function f of order $k+1$	$f(x) := P_i(x)$, if $x \in [x_i, x_{i+1})$, $i = 1,..., n-1$ where • $\{x_i\}_{i= 1,..., n}$ is a strictly increasing sequence of breakpoints. • $P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + ... + c_{i,k}(x - x_i)^k$ is a polynomial of degree k (order $k+1$) over the interval $x \in [x_i, x_{i+1})$.
Function p agrees with function f at the points $\{x_i\}_{i=1,...,n}$.	For every point ζ in sequence $\{x_i\}_{i=1,...,n}$ that occurs m times, the equality $p^{(i-1)}(\zeta) = f^{(i-1)}(\zeta)$ holds for all $i = 1,...,m$, where $p^{(i)}(t)$ is the derivative of the i-th order.
The k-th divided difference of function f at points $x_i,..., x_{i+k}$. This difference is the leading coefficient of the polynomial of order $k+1$ that agrees with f at $x_i,..., x_{i+k}$.	$[x_i,..., x_{i+k}] f$ In particular, • $[x_1]f = f(x_1)$ • $[x_1, x_2] f = (f(x_1) - f(x_2)) / (x_1 - x_2)$
A k-order derivative of interpolant $f(x)$ at interpolation site τ .	$f^{(k)}(\tau)$

Interpolants to the Function f at $x_1,..., x_n$ and Boundary Conditions

Concept	Mathematical Notation
Linear interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i)$, where • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$

Concept	Mathematical Notation
	• $c_{2,i} = [x_i, x_{i+1}]f$ • $i = 1,\ldots, n\text{-}1$
Piecewise parabolic interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2$, $x \in [x_i, x_{i+1})$ Coefficients $c_{1,i}$, $c_{2,i}$, and $c_{3,i}$ depend on the conditions: • $P_i(x_i) = f(x_i)$ • $P_i(x_{i+1}) = f(x_{i+1})$ • $P_i((x_{i+1} + x_i) / 2) = v_{i+1}$ where parameter v_{i+1} depends on the interpolant being continuously differentiable: $P_{i-1}^{(1)}(x_i) = P_i^{(1)}(x_i)$

Piecewise parabolic Subbotin interpolant	$P(x) = P_i(x) = c_{1,i}+c_{2,i}(x-x_i)+c_{3,i}(x-x_i)^2+d_{3,i}((x-t_i)_+)^2$,
	where
	• $x \in [t_i, t_{i+1})$
	• $\{t_i\}_{i=1,\ldots,n+1}$ is a sequence of knots such that
	  • $t_1 = x_1$, $t_{n+1} = x_n$
	  • $t_i \in (x_{i-1}, x_i)$, $i = 2,\ldots, n$
	• $x_+ = f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$
	Coefficients $c_{1,i}$, $c_{2,i}$, $c_{3,i}$, and $d_{3,i}$ depend on the following conditions:
	• $P_i(x_i) = f(x_i)$, $P_i(x_{i+1}) = f(x_{i+1})$
	• $P(x)$ is a continuously differentiable polynomial of the second degree on $[t_i, t_{i+1})$, $i = 1,\ldots, n$.

Piecewise cubic Hermite interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3$, where • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1,\ldots, n\text{-}1$ • $s_i = f^{(1)}(x_i)$
Piecewise cubic Bessel interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3$, where • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1,\ldots, n\text{-}1$ • $s_i = (\Delta x_i[x_{i-1}, x_i]f + \Delta x_{i-1}[x_i, x_{i+1}]f) / (\Delta x_i + \Delta x_{i+1})$
Piecewise cubic Akima interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3$, where • $x \in [x_i, x_{i+1})$

Concept	Mathematical Notation
	• $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1,..., n-1$ • $s_i = (w_{i+1}[x_{i-1}, x_i]f + w_{i-1}[xf_i, x_{i+1}]f) / (w_{i+1} + w_{i-1})$, where $w_i = \lvert[x_i, x_{i+1}]f - [x_{i-1}, x_i]f\rvert$
Piecewise natural cubic interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3$, where • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1,..., n-1$ • Parameter s_i depends on the condition that the interpolant is twice continuously differentiable: $P_{i-1}^{(2)}(x_i) = P_i^{(2)}(x_i)$.
Not-a-knot boundary condition.	Parameters s_1 and s_n provide $P_1 = P_2$ and $P_{n-1} = P_n$, so that the first and the last interior breakpoints are inactive.
Free-end boundary condition.	$f''(x_1) = f''(x_n) = 0$
Look-up interpolator for discrete set of points $(x_1, y_1),..., (x_n, y_n)$.	$$y(x) = \begin{cases} y_1, & if\ x = x_1 \\ y_2, & if\ x = x_2 \\ \cdots & \\ y_n, & if\ x = x_n \\ error, & otherwise \end{cases}$$
Step-wise constant continuous right interpolator.	$$y(x) = \begin{cases} y_1, & \text{if } x_1 \le x < x_2 \\ y_2, & \text{if } x_2 \le x < x_3 \\ \cdots & \\ y_{n-1}, & \text{if } x_{n-1} \le x < x_n \\ y_n, & if\ x = x_n \end{cases}$$
Step-wise constant continuous left interpolator.	$$y(x) = \begin{cases} y_1, & \text{if } x = x_1 \\ y_2, & \text{if } x_1 < x \le x_2 \\ y_3, & \text{if } x_2 < x \le x_3 \\ \cdots & \\ y_n, & \text{if } x_{n-1} < x \le x_n \end{cases}$$

Data Fitting Usage Model

Consider an algorithm that uses the Data Fitting functions. Typically, such algorithms consist of four steps or stages:

1. Create a task. You can call the Data Fitting function several times to create multiple tasks.

```
status = dfdnewtask1d( task, nx, x, xhint, ny, y, yhint );
```

2. Modify the task parameters.

```
status = dfdeditppspline1d( task, s_order, c_type, bc_type, bc, ic_type, ic,
scoeff, scoeffhint );
```

3. Perform Data Fitting spline-based computations. You may reiterate steps 2-3 as needed.

```
status = dfdinterpolate1d(task, estimate, method, nsite, site, sitehint, ndorder,
dorder, datahint, r, rhint, cell );
```

4. Destroy the task or tasks.

```
status = dfdeletetask( task );
```

See Also

Data Fitting Usage Examples

Data Fitting Usage Examples

You can get Fortran source code in the `.\examples\datafittingf`subdirectory of the Intel® oneAPI Math Kernel Library (oneMKL) installation directory.

Data Fitting Function Task Status and Error Reporting

The Data Fitting routines report a task status through integer values. Negative status values indicate errors, while positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The status codes have symbolic names predefined in the header file as integer constants via the `PARAMETER` operators.

If no error occurred, the function returns the `DF_STATUS_OK` code defined as zero:

```
INTEGER, PARAMETER::DF_STATUS_OK = 0
```

In case of an error, the function returns a non-zero error code that specifies the origin of the failure. Header files define the following status codes:

Status Codes in the Data Fitting Component

Status Code	Description
Common Status Codes	
`DF_STATUS_OK`	Operation completed successfully.
`DF_ERROR_NULL_TASK`	Data Fitting task is a `NULL` pointer.
`DF_ERROR_MEM_FAILURE`	Memory allocation failure.
`DF_ERROR_METHOD_NOT_SUPPORTED`	Requested method is not supported.
`DF_ERROR_COMP_TYPE_NOT_SUPPORTED`	Requested computation type is not supported.
`DF_ERROR_NULL_PTR`	Pointer to parameter is null.

Status Code	Description
Data Fitting Task Creation and Initialization, and Generic Editing Operations	
DF_ERROR_BAD_NX	Invalid number of breakpoints.
DF_ERROR_BAD_X	Array of breakpoints is invalid.
DF_ERROR_BAD_X_HINT	Invalid hint describing the structure of the partition.
DF_ERROR_BAD_NY	Invalid dimension of vector-valued function *y*.
DF_ERROR_BAD_Y	Array of function values is invalid.
DF_ERROR_BAD_Y_HINT	Invalid flag describing the structure of function *y*
Data Fitting Task-Specific Editing Operations	
DF_ERROR_BAD_SPLINE_ORDER	Invalid spline order.
DF_ERROR_BAD_SPLINE_TYPE	Invalid spline type.
DF_ERROR_BAD_IC_TYPE	Type of internal conditions used for spline construction is invalid.
DF_ERROR_BAD_IC	Array of internal conditions for spline construction is not defined.
DF_ERROR_BAD_BC_TYPE	Type of boundary conditions used in spline construction is invalid.
DF_ERROR_BAD_BC	Array of boundary conditions for spline construction is not defined.
DF_ERROR_BAD_PP_COEFF	Array of piecewise polynomial spline coefficients is not defined.
DF_ERROR_BAD_PP_COEFF_HINT	Invalid flag describing the structure of the piecewise polynomial spline coefficients.
DF_ERROR_BAD_PERIODIC_VAL	Function values at the endpoints of the interpolation interval are not equal as required in periodic boundary conditions.
DF_ERROR_BAD_DATA_ATTR	Invalid attribute of the pointer to be set or modified in Data Fitting task descriptor with the `df?editidxptr` task editor.
DF_ERROR_BAD_DATA_IDX	Index of the pointer to be set or modified in the Data Fitting task descriptor with the `df?editidxptr` task editor is out of the pre-defined range.
Data Fitting Computation Operations	
DF_ERROR_BAD_NSITE	Invalid number of interpolation sites.
DF_ERROR_BAD_SITE	Array of interpolation sites is not defined.
DF_ERROR_BAD_SITE_HINT	Invalid flag describing the structure of interpolation sites.
DF_ERROR_BAD_NDORDER	Invalid size of the array defining derivative orders to be computed at interpolation sites.

Status Code	Description
DF_ERROR_BAD_DORDER	Array defining derivative orders to be computed at interpolation sites is not defined.
DF_ERROR_BAD_DATA_HINT	Invalid flag providing additional information about partition or interpolation sites.
DF_ERROR_BAD_INTERP	Array of spline-based interpolation results is not defined.
DF_ERROR_BAD_INTERP_HINT	Invalid flag defining the structure of spline-based interpolation results.
DF_ERROR_BAD_CELL_IDX	Array of indices of partition cells containing interpolation sites is not defined.
DF_ERROR_BAD_NLIM	Invalid size of arrays containing integration limits.
DF_ERROR_BAD_LLIM	Array of the left-side integration limits is not defined.
DF_ERROR_BAD_RLIM	Array of the right-side integration limits is not defined.
DF_ERROR_BAD_INTEGR	Array of spline-based integration results is not defined.
DF_ERROR_BAD_INTEGR_HINT	Invalid flag providing the structure of the array of spline-based integration results.
DF_ERROR_BAD_LOOKUP_INTERP_SITE	Bad site provided for interpolation with look-up interpolator.

NOTE
The routine that estimates piecewise polynomial cubic spline coefficients can return internal error codes related to the specifics of the implementation. Such error codes indicate invalid input data or other issues unrelated to Data Fitting routines.

Data Fitting Task Creation and Initialization Routines

Task creation and initialization routines are functions used to create a new task descriptor and initialize its parameters. The Data Fitting component provides the `df?newtask1d` routine that creates and initializes a new task descriptor for a one-dimensional Data Fitting task.

df?newtask1d

Creates and initializes a new task descriptor for a one-dimensional Data Fitting task.

Syntax

status = dfsnewtask1d(*task*, *nx*, *x*, *xhint*, *ny*, *y*, *yhint*)

status = dfdnewtask1d(*task*, *nx*, *x*, *xhint*, *ny*, *y*, *yhint*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
nx	INTEGER	Number of breakpoints representing partition of interpolation interval [a, b].
x	REAL(KIND=4) DIMENSION(*) for dfsnewtask1d REAL(KIND=8) DIMENSION(*) for dfdnewtask1d	One-dimensional array containing the strictly sorted breakpoints from interpolation interval [a, b]. The structure of the array is defined by parameter *xhint*: • If partition is non-uniform or quasi-uniform, the array should contain *nx* strictly ordered values. • If partition is uniform, the array should contain two entries that represent endpoints of interpolation interval [a, b]. **Caution** The array must be strictly sorted. If it is unordered, the results of data fitting routines are not correct.
xhint	INTEGER	A flag describing the structure of partition *x*. For the list of possible values of *xhint*, see table "Hint Values for Partition *x*". If you set the flag to the DF_NO_HINT value, the library interprets the partition as non-uniform.
ny	INTEGER	Dimension of vector-valued function *y*.
y	REAL(KIND=4) DIMENSION(*) for dfsnewtask1d REAL(KIND=8) DIMENSION(*) for dfdnewtask1d	Vector-valued function *y*, array of size *nx***ny*. The storage format of function values in the array is defined by the value of flag *yhint*.
yhint	INTEGER	A flag describing the structure of array *y*. Valid hint values are listed in table "Hint Values for Vector-Valued Function *y*". If you set the flag to the DF_NO_HINT value, the library assumes that all *ny* coordinates of the vector-valued function *y* are provided and stored in row-major format.

Output Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
status	INTEGER	Status of the routine: • DF_STATUS_OK if the task is created successfully. • Non-zero error code if the task creation failed. See "Task Status and Error Reporting" for error code definitions.

Description

The df?newtask1d routine creates and initializes a new Data Fitting task descriptor with user-specified parameters for a one-dimensional Data Fitting task. The *x* and *nx* parameters representing the partition of interpolation interval [*a*, *b*] are mandatory. If you provide invalid values for these parameters, such as a NULL pointer *x* or the number of breakpoints smaller than two, the routine does not create the Data Fitting task and returns an error code.

If you provide a vector-valued function *y*, make sure that the function dimension *ny* and the array of function values *y* are both valid. If any of these parameters are invalid, the routine does not create the Data Fitting task and returns an error code.

If you store coordinates of the vector-valued function *y* in non-contiguous memory locations, you can set the *yhint* flag to DF_1ST_COORDINATE, and pass only the first coordinate of the function into the task creation routine. After successful creation of the Data Fitting task, you can pass the remaining coordinates using the df?editidxptr task editor.

If the routine fails to create the task descriptor, it returns a NULL task pointer.

The routine supports the following hint values for partition *x*:

Hint Values for Partition *x*

Value	Description
DF_NON_UNIFORM_PARTITION	Partition is non-uniform.
DF_QUASI_UNIFORM_PARTITION	Partition is quasi-uniform.
DF_UNIFORM_PARTITION	Partition is uniform.
DF_NO_HINT	No hint is provided. By default, partition is interpreted as non-uniform.

The routine supports the following hint values for the vector-valued function:

Hint Values for Vector-Valued Function *y*

Value	Description
DF_MATRIX_STORAGE_ROWS	Data is stored in row-major format according to C conventions.
DF_MATRIX_STORAGE_COLS	Data is stored in column-major format according to Fortran conventions.
DF_1ST_COORDINATE	The first coordinate of vector-valued data is provided.
DF_NO_HINT	No hint is provided. By default, the coordinates of vector-valued function *y* are provided and stored in row-major format.

NOTE
You must preserve the arrays *x* (breakpoints) and *y* (vector-valued functions) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

Task Configuration Routines

In order to configure tasks, you can use task editors and task query routines.

Task editors initialize or change the predefined Data Fitting task parameters. You can use task editors to initialize or modify pointers to arrays or parameter values.

Task editors can be task-specific or generic. Task-specific editors can modify more than one parameter related to a specific task. Generic editors modify a single parameter at a time.

The Data Fitting component of Intel® oneAPI Math Kernel Library (oneMKL) provides the following task editors:

Data Fitting Task Editors

Editor	Description	Type
df? editppspline1d	Changes parameters of the piecewise polynomial spline.	Task-specific
df?editptr	Changes a pointer in the task descriptor.	Generic
dfieditval	Changes a value in the task descriptor.	Generic
df?editidxptr	Changes a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.	Generic

Task query routines are used to read the predefined Data Fitting task parameters. You can use task query routines to read the values of pointers or parameters.

Task query routines are generic (not task-specific), allowing you to read a single parameter at a time.

The Data Fitting component of the Intel® oneAPI Math Kernel Library (oneMKL) provides the following task query routines:

Data Fitting Task Query Routines

Editor	Description	Type
df?queryptr	Queries a pointer in the task descriptor.	Generic
dfiqueryval	Queries a value in the task descriptor.	Generic
df?queryidxptr	Queries a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.	Generic

df?editppspline1d

Modifies parameters representing a spline in a Data Fitting task descriptor.

Syntax

status = dfseditppspline1d(*task*, *s_order*, *s_type*, *bc_type*, *bc*, *ic_type*, *ic*, *scoeff*, *scoeffhint*)

status = dfdeditppspline1d(*task*, *s_order*, *s_type*, *bc_type*, *bc*, *ic_type*, *ic*, *scoeff*, *scoeffhint*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
s_order	INTEGER	Spline order. The parameter takes one of the values described in table "Spline Orders Supported by Data Fitting Functions".
s_type	INTEGER	Spline type. The parameter takes one of the values described in table "Spline Types Supported by Data Fitting Functions".
bc_type	INTEGER	Type of boundary conditions. The parameter takes one of the values described in table "Boundary Conditions Supported by Data Fitting Functions".
bc	REAL(KIND=4) DIMENSION(*) for dfseditppspline1d REAL(KIND=8) DIMENSION(*) for dfdeditppspline1d	Pointer to boundary conditions. The size of the array is defined by the value of parameter *bc_type*: • If you set free-end or not-a-knot boundary conditions, pass the NULL pointer to this parameter. • If you combine boundary conditions at the endpoints of the interpolation interval, pass an array of two elements. • If you set a boundary condition for the default quadratic spline or a periodic condition for Hermite or the default cubic spline, pass an array of one element.
ic_type	INTEGER	Type of internal conditions. The parameter takes one of the values described in table "Internal Conditions Supported by Data Fitting Functions".
ic	REAL(KIND=4) DIMENSION(*) for dfseditppspline1d REAL(KIND=8) DIMENSION(*) for dfdeditppspline1d	A non-NULL pointer to the array of internal conditions. The size of the array is defined by the value of parameter *ic_type*: • If you set first derivatives or second derivatives internal conditions (*ic_type*=DF_IC_1ST_DER or *ic_type*=DF_IC_2ND_DER), pass an array of n-1 derivative values at the internal points of the interpolation interval. • If you set the knot values internal condition for Subbotin spline (*ic_type*=DF_IC_Q_KNOT) and the knot partition is non-uniform, pass an array of n+1 elements. • If you set the knot values internal condition for Subbotin spline (*ic_type*=DF_IC_Q_KNOT) and the knot partition is uniform, pass an array of four elements.
scoeff	REAL(KIND=4) DIMENSION(*) for dfseditppspline1d REAL(KIND=8) DIMENSION(*) for dfdeditppspline1d	Spline coefficients. An array of size *ny*s_order**(*nx*-1). The storage format of the coefficients in the array is defined by the value of flag *scoeffhint*.

Name	Type	Description
scoeffhint	INTEGER	A flag describing the structure of the array of spline coefficients. For valid hint values, see table "Hint Values for Spline Coefficients". The library stores the coefficients in row-major format. The default value is DF_NO_HINT.

Output Parameters

Name	Type	Description
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Description

The editor modifies parameters that describe the order, type, boundary conditions, internal conditions, and coefficients of a spline. The spline order definition is provided in the "Mathematical Conventions" section. You can set the spline order to any value supported by Data Fitting functions. The table below lists the available values:

Spline Orders Supported by the Data Fitting Functions

Order	Description
DF_PP_STD	Artificial value. Use this value for look-up and step-wise constant interpolants only.
DF_PP_LINEAR	Piecewise polynomial spline of the second order (linear spline).
DF_PP_QUADRATIC	Piecewise polynomial spline of the third order (quadratic spline).
DF_PP_CUBIC	Piecewise polynomial spline of the fourth order (cubic spline).

To perform computations with a spline not supported by Data Fitting routines, set the parameter defining the spline order and pass the spline coefficients to the library in the supported format. For format description, see figure "Row-major Coefficient Storage Format".

The table below lists the supported spline types:

Spline Types Supported by Data Fitting Functions

Type	Description
DF_PP_DEFAULT	The default spline type. You can use this type with linear, quadratic, or user-defined splines.
DF_PP_SUBBOTIN	Quadratic splines based on Subbotin algorithm, [StechSub76].
DF_PP_NATURAL	Natural cubic spline.

Type	Description
DF_PP_HERMITE	Hermite cubic spline.
DF_PP_BESSEL	Bessel cubic spline.
DF_PP_AKIMA	Akima cubic spline.
DF_LOOKUP_INTERPOLANT	Look-up interpolant.
DF_CR_STEPWISE_CONST_INTERPOLANT	Continuous right step-wise constant interpolant.
DF_CL_STEPWISE_CONST_INTERPOLANT	Continuous left step-wise constant interpolant.

If you perform computations with look-up or step-wise constant interpolants, set the spline order to the DF_PP_STD value.

Construction of specific splines may require boundary or internal conditions. To compute coefficients of such splines, you should pass boundary or internal conditions to the library by specifying the type of the conditions and providing the necessary values. For splines that do not require additional conditions, such as linear splines, set condition types to DF_NO_BC and DF_NO_IC, and pass NULL pointers to the conditions. The table below defines the supported boundary conditions:

Boundary Conditions Supported by Data Fitting Functions

Boundary Condition	Description	Spline
DF_NO_BC	No boundary conditions provided.	All
DF_BC_NOT_A_KNOT	Not-a-knot boundary conditions.	Akima, Bessel, Hermite, natural cubic
DF_BC_FREE_END	Free-end boundary conditions.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_1ST_LEFT_DER	The first derivative at the left endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_1ST_RIGHT_DER	The first derivative at the right endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_2ST_LEFT_DER	The second derivative at the left endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_2ND_RIGHT_DER	The second derivative at the right endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_PERIODIC	Periodic boundary conditions.	Linear, all cubic splines
DF_BC_Q_VAL	Function value at point $(x_0 + x_1)/2$	Default quadratic

NOTE
To construct a natural cubic spline, pass these settings to the editor:

- `DF_PP_CUBIC` as the spline order,
- `DF_PP_NATURAL` as the spline type, and
- `DF_BC_FREE_END` as the boundary condition.

To construct a cubic spline with other boundary conditions, pass these settings to the editor:

- `DF_PP_CUBIC` as the spline order,
- `DF_PP_NATURAL` as the spline type, and
- the required type of boundary condition.

For Akima, Hermite, Bessel, and default cubic splines use the corresponding type defined in Table Spline Types Supported by Data Fitting Functions.

You can combine the values of boundary conditions with a bitwise `OR` operation. This permits you to pass combinations of first and second derivatives at the endpoints of the interpolation interval into the library. To pass a first derivative at the left endpoint and a second derivative at the right endpoint, set the boundary conditions to `DF_BC_1ST_LEFT_DER OR DF_BC_2ND_RIGHT_DER`.

You should pass the combined boundary conditions as an array of two elements. The first entry of the array contains the value of the boundary condition for the left endpoint of the interpolation interval, and the second entry - for the right endpoint. Pass other boundary conditions as arrays of one element.

For the conditions defined as a combination of valid values, the library applies the following rules to identify the boundary condition type:

- If not required for spline construction, the value of boundary conditions is ignored.
- Not-a-knot condition has the highest priority. If set, other boundary conditions are ignored.
- Free-end condition has the second priority after the not-a-knot condition. If set, other boundary conditions are ignored.
- Periodic boundary condition has the next priority after the free-end condition.
- The first derivative has higher priority than the second derivative at the right and left endpoints.

If you set the periodic boundary condition, make sure that function values at the endpoints of the interpolation interval are identical. Otherwise, the library returns an error code. The table below specifies the values to be provided for each type of spline if the periodic boundary condition is set.

Boundary Requirements for Periodic Conditions

Spline Type	Periodic Boundary Condition Support	Boundary Value
Linear	Yes	Not required
Default quadratic	No	
Subbotin quadratic	No	
Natural cubic	Yes	Not required
Bessel	Yes	Not required
Akima	Yes	Not required
Hermite cubic	Yes	First derivative
Default cubic	Yes	Second derivative

Internal conditions supported in the Data Fitting domain that you can use for the *ic_type* parameter are the following:

Internal Conditions Supported by Data Fitting Functions

Internal Condition	Description	Spline
DF_NO_IC	No internal conditions provided.	
DF_IC_1ST_DER	Array of first derivatives of size n-2, where n is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.	Hermite cubic
DF_IC_2ND_DER	Array of second derivatives of size n-2, where n is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.	Default cubic
DF_IC_Q_KNOT	Knot array of size n+1, where n is the number of breakpoints.	Subbotin quadratic

To construct a Subbotin quadratic spline, you have three options to get the array of knots in the library:

- If you do not provide the knots, the library uses the default values of knots $t = \{t_i\}$, $i = 0, ..., n$ according to the rule:

  $t_0 = x_0$, $t_n = x_{n-1}$, $t_i = (x_i + x_{i-1})/2$, $i = 1, ..., n - 1$.
- If you provide the knots in an array of size $n + 1$, the knots form a non-uniform partition. Make sure that the knot values you provide meet the following conditions:

  $t_0 = x_0$, $t_n = x_{n-1}$, $t_i \in (x_{i-1}, x_i)$, $i = 1,..., n - 1$.
- If you provide the knots in an array of size 4, the knots form a uniform partition

  $t_0 = x_0$, $t_1 = l$, $t_2 = r$, $t_3 = x_{n-1}$, where $l \in (x_0, x_1)$ and $r \in (x_{n-2}, x_{n-1})$.

  In this case, you need to set the value of the *ic_type* parameter holding the type of internal conditions to DF_IC_Q_KNOT OR DF_UNIFORM_PARTITION.

> **NOTE**
> Since the partition is uniform, perform an OR operation with the DF_UNIFORM_PARTITION partition hint value described in Table Hint Values for Partition x.

For computations based on look-up and step-wise constant interpolants, you can avoid calling the df?editppspline1d editor and directly call one of the routines for spline-based computation of spline values, derivatives, or integrals. For example, you can call the df?construct1d routine to construct the required spline with the given attributes, such as order or type.

The memory location of the spline coefficients is defined by the *scoeff* parameter. Make sure that the size of the array is sufficient to hold *ny***s_order* * (*nx*-1) values.

The df?editppspline1d routine supports the following hint values for spline coefficients:

Hint Values for Spline Coefficients

Order	Description
DF_1ST_COORDINATE	The first coordinate of vector-valued data is provided.
DF_NO_HINT	No hint is provided. By default, all sets of spline coefficients are stored in row-major format.

The coefficients for all coordinates of the vector-valued function are packed in memory one by one in successive order, from function y_1 to function y_{ny}.

Within each coordinate, the library stores the coefficients as an array, in row-major format:

$c_{1,0}$, $c_{1,1}$, ..., $c_{1,k}$, $c_{2,0}$, $c_{2,1}$, ..., $c_{2,k}$, ..., $c_{n-1,0}$, $c_{n-1,1}$, ..., $c_{n-1,k}$

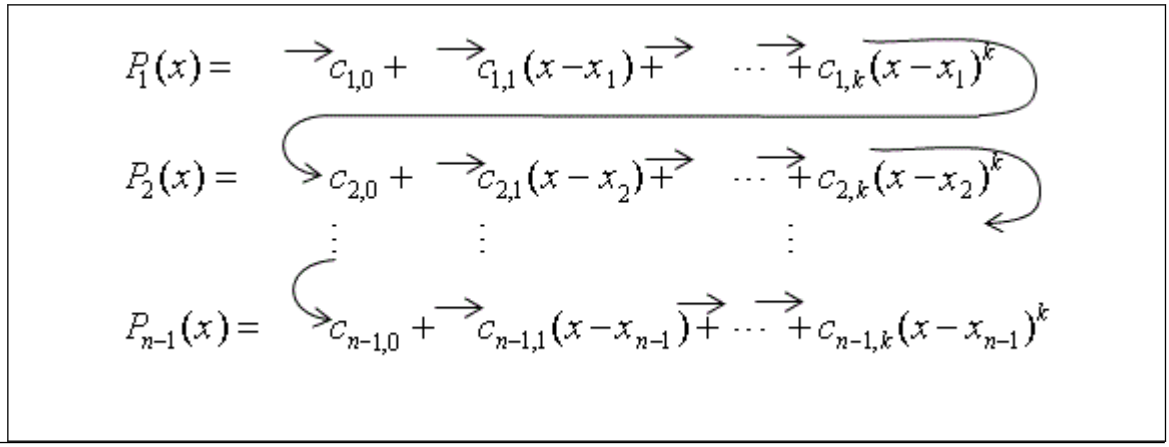
Mapping of the coefficients to storage in the *scoeff* array is described below, where $c_{i,j}$ is the jth coefficient of the function

$$P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + \ldots + c_{i,k}(x - x_i)^k$$

.

See Mathematical Conventions for more details on nomenclature and interpolants.

Row-major Coefficient Storage Format



If you store splines corresponding to different coordinates of the vector-valued function at non-contiguous memory locations, do the following:

1. Set the *scoeffhint* flag to DF_1ST_COORDINATE and provide the spline for the first coordinate.
2. Pass the spline coefficients for the remaining coordinates into the Data Fitting task using the df?editidxptr task editor.

Using the df?editppspline1d task editor, you can provide to the Data Fitting task an already constructed spline that you want to use in computations. To ensure correct interpretation of the memory content, you should set the following parameters:

- Spline order and type, if appropriate. If the spline is not supported by the library, set the *s_type* parameter to DF_PP_DEFAULT.
- Pointer to the array of spline coefficients in row-major format.
- The *scoeffhint* parameter describing the structure of the array:
  - Set the *scoeffhint* flag to the DF_1ST_COORDINATE value to pass spline coefficients stored at different memory locations. In this case, you can set the parameters that describe boundary and internal conditions to zero.

- Use the default value DF_NO_HINT for all other cases.

Before passing an already constructed spline into the library, you should call the dfieditval task editor to provide the dimension of the spline DF_NY. See table "Parameters Supported by the dfieditval Task Editor" for details.

After you provide the spline to the Data Fitting task, you can run computations that use this spline.

NOTE
You must preserve the arrays *bc* (boundary conditions), *ic* (internal conditions), and *scoeff* (spline coefficients) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

df?editptr

Modifies a pointer to an array held in a Data Fitting task descriptor.

Syntax

status = dfseditptr(*task*, *ptr_attr*, *ptr*)

status = dfdeditptr(*task*, *ptr_attr*, *ptr*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
ptr_attr	INTEGER	The parameter to change. For details, see the *Pointer Attribute* column in table "Pointers Supported by the df?editptr Task Editor".
ptr	REAL(KIND=4) DIMENSION(*) for dfseditptr REAL(KIND=8) DIMENSION(*) for dfdeditptr	New pointer. For details, see the *Purpose* column in table "Pointers Supported by the df?editptr Task Editor".

Output Parameters

Name	Type	Description
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The df?editptr editor replaces the pointer of type *ptr_attr* stored in a Data Fitting task descriptor with a new pointer *ptr*. The table below describes types of pointers supported by the editor:

Pointers Supported by the df?editptr Task Editor

Pointer Attribute	Purpose
DF_X	Partition *x* of the interpolation interval, an array of strictly sorted breakpoints. **Caution** The array must be strictly sorted. If it is unordered, the results of data fitting routines are not correct.
DF_Y	Vector-valued function *y*
DF_IC	Internal conditions for spline construction. For details, see table "Internal Conditions Supported by Data Fitting Functions".
DF_BC	Boundary conditions for spline construction. For details, see table "Boundary Conditions Supported by Data Fitting Functions".
DF_PP_SCOEFF	Spline coefficients

You can use df?editptr to modify different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory. Use the df?editidxptr editor if you need to modify pointers to coordinates of the vector-valued function or spline coefficients stored at non-contiguous memory locations.

If you modify a partition of the interpolation interval, then you should call the dfieditval task editor with the corresponding value of DF_XHINT, even if the structure of the partition remains the same.

If you pass a NULL pointer to the df?editptr task editor, the task remains unchanged and the routine returns an error code. For the predefined error codes, please see "Task Status and Error Reporting".

NOTE
You must preserve the arrays *x* (breakpoints), *y* (vector-valued functions), *bc* (boundary conditions), *ic* (internal conditions), and *scoeff* (spline coefficients) through the entire workflow of the Data Fitting computations which use those arrays, as the task stores the addresses of the arrays for spline-based computations.

dfieditval

Modifies a parameter value in a Data Fitting task descriptor.

Syntax

status = dfieditval(*task*, *val_attr*, *val*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.

Name	Type	Description
val_attr	INTEGER	The parameter to change. See table "Parameters Supported by the dfieditval Task Editor".
val	INTEGER	A new parameter value. See table "Parameters Supported by the dfieditval Task Editor".

Output Parameters

Name	Type	Description
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The dfieditval task editor replaces the parameter of type *val_attr* stored in a Data Fitting task descriptor with a new value *val*. The table below describes valid types of parameter *val_attr* supported by the editor:

Parameters Supported by the dfieditval Task Editor

Parameter Attribute	Purpose
DF_NX	Number of breakpoints
DF_XHINT	A flag describing the structure of partition. See table "Hint Values for Partition *x*" for the list of available values.
DF_NY	Dimension of the vector-valued function
DF_YHINT	A flag describing the structure of the vector-valued function. See table "Hint Values for Vector Function *y*" for the list of available values.
DF_SPLINE_ORDER	Spline order. See table "Spline Orders Supported by Data Fitting Functions" for the list of available values.
DF_SPLINE_TYPE	Spline type. See table "Spline Types Supported by Data Fitting Functions" for the list of available values.
DF_BC_TYPE	Type of boundary conditions used in spline construction. See table "Boundary Conditions Supported by Data Fitting Functions" for the list of available values.
DF_IC_TYPE	Type of internal conditions used in spline construction. See table "Internal Conditions Supported by Data Fitting Functions" for the list of available values.
DF_PP_COEFF_HINT	A flag describing the structure of spline coefficients. See table "Hint Values for Spline Coefficients" for the list of available values.
DF_CHECK_FLAG	A flag which controls checking of Data Fitting parameters. See table "Possible Values for the DF_CHECK_FLAG Parameter" for the list of available values.

If you pass a zero value for the parameter describing the size of the arrays that hold coefficients for a partition, a vector-valued function, or a spline, the parameter held in the Data fitting task remains unchanged and the routine returns an error code. For the predefined error codes, see "Task Status and Error Reporting".

Possible Values for the DF_CHECK_FLAG Parameter

Value	Description
DF_ENABLE_CHECK_FLAG	Checks the correctness of parameters of Data Fitting computational routines (default mode).
DF_DISABLE_CHECK_FLAG	Disables checking of the correctness of parameters of Data Fitting computational routines.

Use DF_CHECK_FLAG for *val_attr* in order to control validation of parameters of Data Fitting computational routines such as df?construct1d, df?interpolate1d/df?interpolateex1d, and df?searchcells1d/df?searchcellsex1d, which can perform better with a small number of interpolation sites or integration limits (fewer than one dozen). The default mode, with checking of parameters enabled, should be used as you develop a Data Fitting-based application. After you complete development you can disable parameter checking in order to improve the performance of your application.

If you modify the parameter describing dimensions of the arrays that hold the vector-valued function or spline coefficients in contiguous memory, you should call the df?editptr task editor with the corresponding pointers to the vector-valued function or spline coefficients even when this pointer remains unchanged. Call the df?editidxptr editor if those arrays are stored in non-contiguous memory locations.

You must call the dfieditval task editor to edit the structure of the partition DF_XHINT every time you modify a partition using df?editptr, even if the structure of the partition remains the same.

df?editidxptr

Modifies a pointer to the memory representing a coordinate of the data stored in matrix format.

Syntax

status = dfseditidxptr(*task*, *ptr_attr*, *idx*, *ptr*)

status = dfdeditidxptr(*task*, *ptr_attr*, *idx*, *ptr*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
ptr_attr	INTEGER	Type of the data to be modified. The parameter takes one of the values described in "Data Attributes Supported by the df?editidxptr Task Editor".
idx	INTEGER	Index of the coordinate whose pointer is to be modified.
ptr	REAL(KIND=4) DIMENSION(*) for dfseditidxptr	Pointer to the data that holds values of coordinate *idx*. For details, see table "Data Attributes Supported by the df?editidxptr Task Editor".

Name	Type	Description
	REAL(KIND=8) DIMENSION(*) for dfdeditidxptr	

Output Parameters

Name	Type	Description
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The routine modifies a pointer to the array that holds the *idx* coordinate of vector-valued function *y* or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the editor if you need to pass into a Data Fitting task or modify the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations. Do not use the editor for coordinates at contiguous memory locations in row-major format.

Before calling this editor, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific df?editppspline1d editor.

Data Attributes Supported by the df?editidxptr Task Editor

Data Attribute	Description
DF_Y	Vector-valued function *y*
DF_PP_SCOEFF	Piecewise polynomial spline coefficients

When using df?editidxptr, you might receive an error code in the following cases:

- You passed an unsupported parameter value into the editor.
- The value of the index exceeds the predefined value that equals the dimension *ny* of the vector-valued function.
- You pass a NULL pointer to the editor. In this case, the task remains unchanged.
- You pass a pointer to the *idx* coordinate of the vector-valued function you provided to contiguous memory in column-major format.

The code example below demonstrates how to use the editor for providing values of a vector-valued function stored in two non-contiguous arrays:

df?queryptr

Reads a pointer to an array held in a Data Fitting task descriptor.

Syntax

status = dfsqueryptr(*task*, *ptr_attr*, *ptr*)

status = dfdqueryptr(*task*, *ptr_attr*, *ptr*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
ptr_attr	INTEGER	The parameter to query. The query routine supports pointer attributes described in the table "Pointers Supported by the df?editptr Task Editor". For details, see the *Pointer Attribute* column in the table.

Output Parameters

Name	Type	Description
ptr	INTEGER(KIND=8)	Pointer to array returned by the query routine. For details, see the *Purpose* column in table "Pointers Supported by the df?editptr Task Editor".
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The df?queryptr routine returns the pointer of type *ptr_attr* stored in a Data Fitting task descriptor as parameter *ptr*. Attributes of the pointers supported by the query function are identical to those supported by the editor df?editptr editor in the table "Pointers Supported by the df?editptr Task Editor".

You can use df?queryptr to read different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory.

dfiqueryval

Reads a parameter value in a Data Fitting task descriptor.

Syntax

status = dfiqueryval(*task*, *val_attr*, *val*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.

Name	Type	Description
val_attr	INTEGER	The parameter to query. The query function supports the parameter attributes described in "Parameters Supported by the dfieditval Task Editor".

Output Parameters

Name	Type	Description
val	INTEGER	The parameter value returned by the query function. See table "Parameters Supported by the dfieditval Task Editor".
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The dfiqueryval routine returns a parameter of type *val_attr* stored in a Data Fitting task descriptor as parameter *val*. The query function supports the parameter attributes described in "Parameters Supported by the dfieditval Task Editor".

df?queryidxptr

Reads a pointer to the memory representing a coordinate of the data stored in matrix format.

Syntax

status = dfsqueryidxptr(*task*, *ptr_attr*, *idx*, *ptr*)

status = dfdqueryidxptr(*task*, *ptr_attr*, *idx*, *ptr*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
ptr_attr	INTEGER	Pointer attribute to query. The parameter takes one of the attributes described in "Data Attributes Supported by the df?editidxptr Task Editor".
idx	INTEGER	Index of the coordinate of the pointer to query.

Output Parameters

Name	Type	Description
ptr	INTEGER(KIND=8)	Pointer to the data that holds values of coordinate *idx* returned. For details, see table "Data Attributes Supported by the df?editidxptr Task Editor".
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The routine returns a pointer to the array that holds the *idx* coordinate of vector-valued function *y* or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the query routine if you need the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations or at a contiguous memory location in row-major format (the default storage format for spline coefficients).

Before calling this query routine, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific df?editppspline1d editor.

When using df?queryidxptr, you might receive an error code in the following cases:

- You passed an unsupported parameter value into the editor.
- The value of the index exceeds the predefined value that equals the dimension *ny* of the vector-valued function.
- You request the pointer to the *idx* coordinate of the vector-valued function you provided to contiguous memory in column-major format.

Data Fitting Computational Routines

Data Fitting computational routines are functions used to perform spline-based computations, such as:

- spline construction
- computation of values, derivatives, and integrals of the predefined order
- cell search

Once you create a Data Fitting task and initialize the required parameters, you can call computational routines as many times as necessary.

The table below lists the available computational routines:

Data Fitting Computational Routines

Routine	Description
df?construct1d	Constructs a spline for a one-dimensional Data Fitting task.
df?interpolate1d	Computes spline values and derivatives.
df?interpolateex1d	Computes spline values and derivatives by calling user-provided interpolants.
df?integrate1d	Computes spline-based integrals.

Routine	Description
df?integrateex1d	Computes spline-based integrals by calling user-provided integrators.
df?searchcells1d	Finds indices of cells containing interpolation sites.
df?searchcellsex1d	Finds indices of cells containing interpolation sites by calling user-provided cell searchers.

If a Data Fitting computation completes successfully, the computational routines return the DF_STATUS_OK code. If an error occurs, the routines return an error code specifying the origin of the failure. Some possible errors are the following:

- The task pointer is NULL.
- Memory allocation failed.
- The computation failed for another reason.

For the list of available status codes, see "Task Status and Error Reporting".

NOTE
Data Fitting computational routines do not control errors for floating-point conditions, such as overflow, gradual underflow, or operations with Not a Number (NaN) values.

df?construct1d

Syntax

Constructs a spline of the given type.

status = dfsconstruct1d(*task*, *s_format*, *method*)

status = dfdconstruct1d(*task*, *s_format*, *method*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
s_format	INTEGER	Spline format. The supported value is DF_PP_SPLINE.
method	INTEGER	Construction method. The supported value is DF_METHOD_STD.

Output Parameters

Name	Type	Description
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Description

Before calling df?construct1d, you need to create and initialize the task, and set the parameters representing the spline. Then you can call the df?construct1d routine to construct the spline. The format of the spline is defined by parameter *s_format*. The method for spline construction is defined by parameter *method*. Upon successful construction, the spline coefficients are available in the user-provided memory location in the format you set through the Data Fitting editor. For the available storage formats, see table "Hint Values for Spline Coefficients".

df?interpolate1d/df?interpolateex1d

Runs data fitting computations.

Syntax

status = dfsinterpolate1d(*task*, *type*, *method*, *nsite*, *site*, *sitehint*, *ndorder*, *dorder*, *datahint*, *r*, *rhint*, *cell*)

status = dfdinterpolate1d(*task*, *type*, *method*, *nsite*, *site*, *sitehint*, *ndorder*, *dorder*, *datahint*, *r*, *rhint*, *cell*)

status = dfsinterpolateex1d(*task*, *type*, *method*, *nsite*, *site*, *sitehint*, *ndorder*, *dorder*, *datahint*, *r*, *rhint*, *cell*, *le_cb*, *le_params*, *re_cb*, *re_params*, *i_cb*, *i_params*, *search_cb*, *search_params*)

status = dfdinterpolateex1d(*task*, *type*, *method*, *nsite*, *site*, *sitehint*, *ndorder*, *dorder*, *datahint*, *r*, *rhint*, *cell*, *le_cb*, *le_params*, *re_cb*, *re_params*, *i_cb*, *i_params*, *search_cb*, *search_params*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
type	INTEGER	Type of spline-based computations. The parameter takes one or more values combined with an OR operation. For the list of possible values, see table "Computation Types Supported by the df?interpolate1d/ df?interpolate1d Routines".
method	INTEGER	Computation method. The supported value is DF_METHOD_PP.
nsite	INTEGER	Number of interpolation sites.
site	REAL(KIND=4) DIMENSION(*) for dfsinterpolate1d/ dfsinterpolateex1d	Array of interpolation sites of size *nsite*. The structure of the array is defined by the *sitehint* parameter: • If sites form a non-uniform partition, the array should contain *nsite* values.

Name	Type	Description
	REAL(KIND=8) DIMENSION(*) for dfdinterpolate1d/ dfdinterpolateex1d	• If sites form a uniform partition, the array should contain two entries that represent the left and the right interpolation sites. The first entry of the array contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point.
sitehint	INTEGER	A flag describing the structure of the interpolation sites. For the list of possible values of *sitehint*, see table "Hint Values for Interpolation Sites". If you set the flag to DF_NO_HINT, the library interprets the site-defined partition as non-uniform.
ndorder	INTEGER	Maximal derivative order increased by one to be computed at interpolation sites.
dorder	INTEGER DIMENSION(*)	Array of size *ndorder* that defines the order of the derivatives to be computed at the interpolation sites. If all the elements in *dorder* are zero, the library computes the spline values only. If you do not need interpolation computations, set *ndorder* to zero and pass a NULL pointer to *dorder*.
datahint	REAL(KIND=4) DIMENSION(*) for dfsinterpolate1d/ dfsinterpolateex1d REAL(KIND=8) DIMENSION(*) for dfdinterpolate1d/ dfdinterpolateex1d	Array that contains additional information about the structure of partition *x* and interpolation sites. This data helps to speed up the computation. If you provide a NULL pointer, the routine uses the default settings for computations. For details on the *datahint* array, see table "Structure of the *datahint* Array".
r	REAL(KIND=4) DIMENSION(*) for dfsinterpolate1d/ dfsinterpolateex1d REAL(KIND=8) DIMENSION(*) for dfdinterpolate1d/ dfdinterpolateex1d	Array for results. If you do not need spline-based interpolation, set this pointer to NULL.
rhint	INTEGER	A flag describing the structure of the results. For the list of possible values of *rhint*, see table "Hint Values for the *rhint* Parameter". If you set the flag to DF_NO_HINT, the library stores the result in row-major format.
cell	INTEGER DIMENSION(*)	Array of cell indices in partition *x* that contain the interpolation sites. Provide this parameter as input if *type* is DF_INTERP_USER_CELL. If you do not need cell indices, set this parameter to NULL.
le_cb	INTEGER	User-defined callback function for extrapolation at the sites to the left of the interpolation interval.

Name	Type	Description
le_params	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the *le_cb* function.
re_cb	INTEGER	User-defined callback function for extrapolation at the sites to the right of the interpolation interval.
re_params	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the *re_cb* function.
i_cb	INTEGER	User-defined callback function for interpolation within the interpolation interval.
i_params	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the *i_cb* function.
search_cb	INTEGER	User-defined callback function for computing indices of cells that can contain interpolation sites.
search_params	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the *search_cb* function.

Output Parameters

Name	Type	Description
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
r		Contains results of computations at the interpolation sites.
cell		Array of cell indices in partition *x* that contain the interpolation sites, which is computed if *type* is DF_CELL.

Description

The df?interpolate1d/df?interpolateex1d routine performs spline-based computations with user-defined settings. The routine supports two types of computations for interpolation sites provided in array *site*:

Computation Types Supported by the df?interpolate1d/df?interpolateex1d Routines

Type	Description
DF_INTERP	Compute derivatives of predefined order. The derivative of the zero order is the spline value.
DF_INTERP_USER_CELL	Compute derivatives of predefined order given user-provided cell indices. The derivative of the zero order is the spline value.

Type	Description
	For this type of the computations you should provide a valid *cell* array, which holds the indices of cells in the *site* array containing relevant interpolation sites.
DF_CELL	Compute indices of cells in partition *x* that contain the sites.

If the indices of cells which contain interpolation types are available before the call to df?interpolate1d/ df?interpolateex1d, you can improve performance by using the DF_INTERP_USER_CELL computation type.

NOTE
If you pass any combination of DF_INTERP, DF_INTERP_USER_CELL, and DF_CELL computation types to the routine, the library uses the DF_INTERP_USER_CELL computation mode.

If you specify DF_INTERP_USER_CELL computation mode and a user-defined callback function for computing cell indices to df?interpolateex1d, the library uses the DF_INTERP_USER_CELL computation mode, and the call-back function is not called.

If the sites do not belong to interpolation interval [*a*, *b*] , the library uses:

- polynomial P_0 of the spline constructed on interval $[x_0, x_1]$ for computations at the sites to the left of *a*.
- polynomial P_{n-2} of the spline constructed on interval $[x_{n-2}, x_{n-1}]$ for computations at the sites to the right of *b*.

Interpolation sites support the following hints:

Hint Values for Interpolation Sites

Value	Description
DF_NON_UNIFORM_PARTITION	Partition is non-uniform.
DF_UNIFORM_PARTITION	Partition is uniform.
DF_SORTED_DATA	Interpolation sites are sorted in the ascending order and define a non-uniform partition.
DF_NO_HINT	No hint is provided. By default, the partition defined by interpolation sites is interpreted as non-uniform.

NOTE
If you pass a sorted array of interpolation sites to the Intel® oneAPI Math Kernel Library (oneMKL), set the*sitehint* parameter to the DF_SORTED_DATA value. The library uses this information when choosing the search algorithm and ignores any other data hints about the structure of the interpolation sites.

Data Fitting computation routines can use the following hints to speed up the computation:

- DF_UNIFORM_PARTITION describes the structure of breakpoints and the interpolation sites.
- DF_QUASI_UNIFORM_PARTITION describes the structure of breakpoints.

Pass the above hints to the library when appropriate.

For spline-based interpolation, you should set the derivatives whose values are required for the computation. You can provide the derivatives by setting the `dorder` array of size `ndorder` as follows:

$$dorder(i) = \begin{cases} 1, \text{ if derivative of the order } i-1 \text{ is required} \\ 0, \text{ otherwise} \end{cases} \quad i = 1, ..., ndorder$$

Orders of derivatives i_d(d = 1, 2, .., *nder*), corresponding to non-zero derivatives to be calculated, form the array $\{i_d\}$ of length *nder*≤`ndorder`.

The storage format for the interpolation results is specified using the `rhint` parameter values. For each storage format, Table Hint Values for the `rhint` Parameter describes how to get the result $R(j, s, i_d)$ from array `r`, for function index j(1 ≤j≤*yn*), site number s(1 ≤s≤*nsite*), and derivative index i_d(1 ≤d≤*nder*), where *yn* is the number of functions, *nsite* is the number of sites, and *nder* is the total number of non-zero derivatives for interpolation. The array `r` can be either a one-dimensional array of size *ny***nder**`nsite` or a three-dimensional array with the dimensions described in the table.

Hint Values for the `rhint` Parameter

Value	Location of $R(j, s, i_d)$, One-dimensional Array Storage	Location of $R(j, s, i_d)$, Three-dimensional Array Storage
`DF_MATRIX_STORAGE_FUNCS_SITES_DERS` (`DF_MATRIX_STORAGE_ROWS`)	`r(d - 1 + nder*(s - 1 + nsite*(j - 1)) + 1)`	`r(d, s, j)` `r` declared as `r(nder, nsite, ny)`.
`DF_MATRIX_STORAGE_FUNCS_DERS_SITES` (`DF_MATRIX_STORAGE_COLS`)	`r(s - 1 + nsite*(d - 1 + nder*(j - 1)) + 1)`	`r(s, d, j)` `r` declared as `r(nsite, nder, ny)`.
`DF_MATRIX_STORAGE_SITES_FUNCS_DERS`	`r(d - 1 + nder*(j - 1 + ny*(s - 1)) + 1)`	`r(d, j, s)` `r` declared as `r(nder, ny, nsite)`.
`DF_MATRIX_STORAGE_SITES_DERS_FUNCS`	`r(j - 1 + ny*(d - 1 + nder*(s - 1)) + 1)`	`r(j, d, s)` `r` declared as `r(ny, nder, nsite)`.
`DF_NO_HINT`	No hint is provided. By default, the results are stored as in `rhint` = `DF_MATRIX_STORAGE_FUNCS_SITES_DERS`.	

The following figures show the structure of the storage formats. Each shows sequential memory layout line by line, left to right.

- Storage in `r` for `rhint` = `DF_MATRIX_STORAGE_FUNCS_SITES_DERS` (`DF_MATRIX_STORAGE_ROWS`):

$$\begin{array}{cccc|cccc|c} R(1,1,i_1) & R(1,2,i_1) & \dots & R(1,nsite,i_1) & R(2,1,i_1) & R(2,2,i_1) & \dots & R(2,nsite,i_1) & \dots \\ R(1,1,i_2) & R(1,2,i_2) & \dots & R(1,nsite,i_2) & R(2,1,i_2) & R(2,2,i_2) & \dots & R(2,nsite,i_2) & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ R(1,1,i_{nder}) & R(1,2,i_{nder}) & \dots & R(1,nsite,i_{nder}) & R(2,1,i_{nder}) & R(2,2,i_{nder}) & \dots & R(2,nsite,i_{nder}) & \dots \end{array}$$

- Storage in `r` for `rhint` = `DF_MATRIX_STORAGE_FUNCS_DERS_SITES` (`DF_MATRIX_STORAGE_COLS`):

$R(1,1,i_1)$ $R(1,1,i_2)$ ... $R(1,1,i_{nder})$ $R(1,2,i_1)$ $R(1,2,i_2)$ ... $R(1,2,i_{nder})$ ... ... $R(1,nsite,i_1)$ $R(1,nsite,i_2)$ ... $R(1,nsite,i_{nder})$	$R(2,1,i_1)$ $R(2,1,i_2)$ ... $R(2,1,i_{nder})$ $R(2,2,i_1)$ $R(2,2,i_2)$ ... $R(2,2,i_{nder})$ ... ... $R(2,nsite,i_1)$ $R(2,nsite,i_2)$ ... $R(2,nsite,i_{nder})$	... ...

- Storage in `r` for `rhint = DF_MATRIX_STORAGE_SITES_FUNCS_DERS`:

$R(1,1,i_1)$ $R(2,1,i_1)$ ... $R(ny,1,i_1)$ $R(1,1,i_2)$ $R(2,1,i_2)$ ... $R(ny,1,i_2)$ ... ... $R(1,1,i_{nder})$ $R(2,1,i_{nder})$ ... $R(ny,1,i_{nder})$	$R(1,2,i_1)$ $R(2,2,i_1)$ ... $R(ny,2,i_1)$ $R(1,2,i_2)$ $R(2,2,i_2)$ ... $R(ny,2,i_2)$ ... ... $R(1,2,i_{nder})$ $R(2,2,i_{nder})$ ... $R(ny,2,i_{nder})$	... ...

- Storage in `r` for `rhint = DF_MATRIX_STORAGE_SITES_DERS_FUNCS`:

$R(1,1,i_1)$ $R(1,1,i_2)$ ... $R(1,1,i_{nder})$ $R(2,1,i_1)$ $R(2,1,i_2)$ ... $R(2,1,i_{nder})$ ... ... $R(ny,1,i_1)$ $R(ny,1,i_2)$ ... $R(ny,1,i_{nder})$	$R(1,2,i_1)$ $R(1,2,i_2)$ ... $R(1,2,i_{nder})$ $R(2,2,i_1)$ $R(2,2,i_2)$ ... $R(2,2,i_{nder})$ ... ... $R(ny,2,i_1)$ $R(ny,2,i_2)$ ... $R(ny,2,i_{nder})$	... ...

To speed up Data Fitting computations, use the `datahint` parameter that provides additional information about the structure of the partition and interpolation sites. This data represents a floating-point or a double array with the following structure:

Structure of the `datahint` Array

Element Number	Description
1	Task dimension
2	Type of additional information
3	Reserved field
4	The total number *q* of elements containing additional information.
5	Element (1)
...	...
q+4	Element (*q*)

Data Fitting computation functions support the following types of additional information for `datahint(2)`:

Types of Additional Information

Type	Element Number	Parameter
`DF_NO_APRIORI_INFO`	0	No parameters are provided. Information about the data structure is absent.

Type	Element Number	Parameter
DF_APRIORI_MOST_LIKELY_CELL	1	Index of the cell that is likely to contain interpolation sites.

To compute indices of the cells that contain interpolation sites, provide the pointer to the array of size *nsite* for the results. The library supports the following scheme of cell indexing for the given partition$\{x_i\}$, i=1,...,nx:

cell(*j*) = i, if *site*(*j*) $\in[x_i, x_{i+1})$, i = 0,..., *nx* - 2,

cell(*j*) = *nx* - 1, if *site*(*j*) $\in[x_{nx-1}, x_{nx}]$,

cell(*j*) = *nx*, if *site*(*j*) $\in(x_{nx}, x_{nx+1}]$,

where

- $x_0 = -\infty$
- $x_{nx+1} = +\infty$
- j = 1,..., *nsite*

To perform interpolation computations with spline types unsupported in the Data Fitting component, use the extended version of the routine df?interpolateex1d. With this routine, you can provide user-defined callback functions for computations within, to the left of, or to the right of interpolaton interval [a, b]. The callback functions compute indices of the cells that contain the specified interpolation sites or can serve as an approximation for computing the exact indices of such cells.

If you do not pass any function for computations at the sites outside the interval [a, b], the routine uses the default settings.

See Also

Mathematical Conventions for Data Fitting Functions

df?interpcallback

df?searchcellscallback

df?integrate1d/df?integrateex1d

Computes a spline-based integral.

Syntax

status = dfsintegrate1d(*task*, *method*, *nlim*, *llim*, *llimhint*, *rlim*, *rlimhint*, *ldatahint*, *rdatahint*, *r*, *rhint*)

status = dfdintegrate1d(*task*, *method*, *nlim*, *llim*, *llimhint*, *rlim*, *rlimhint*, *ldatahint*, *rdatahint*, *r*, *rhint*)

status = dfsintegrateex1d(*task*, *method*, *nlim*, *llim*, *llimhint*, *rlim*, *rlimhint*, *ldatahint*, *rdatahint*, *r*, *rhint*, *le_cb*, *le_params*, *re_cb*, *re_params*, *i_cb*, *i_params*, *search_cb*, *search_params*)

status = dfdintegrateex1d(*task*, *method*, *nlim*, *llim*, *llimhint*, *rlim*, *rlimhint*, *ldatahint*, *rdatahint*, *r*, *rhint*, *le_cb*, *le_params*, *re_cb*, *re_params*, *i_cb*, *i_params*, *search_cb*, *search_params*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
method	INTEGER	Integration method. The supported value is DF_METHOD_PP.
nlim	INTEGER	Number of pairs of integration limits.
llim	REAL(KIND=4) DIMENSION(*) for dfsintegrate1d/ dfsintegrateex1d REAL(KIND=8) DIMENSION(*) for dfdintegrate1d/ dfdintegrateex1d	Array of size *nlim* that defines the left-side integration limits.
llimhint	INTEGER	A flag describing the structure of the left-side integration limits *llim*. For the list of possible values of *llimhint*, see table "Hint Values for Integration Limits". If you set the flag to the DF_NO_HINT value, the library assumes that the left-side integration limits define a non-uniform partition.
rlim	REAL(KIND=4) DIMENSION(*) for dfsintegrate1d/ dfsintegrateex1d REAL(KIND=8) DIMENSION(*) for dfdintegrate1d/ dfdintegrateex1d	Array of size *nlim* that defines the right-side integration limits.
rlimhint	INTEGER	A flag describing the structure of the right-side integration limits *rlim*. For the list of possible values of *rlimhint*, see table "Hint Values for Integration Limits". If you set the flag to the DF_NO_HINT value, the library assumes that the right-side integration limits define a non-uniform partition.
ldatahint	REAL(KIND=4) DIMENSION(*) for dfsintegrate1d/ dfsintegrateex1d REAL(KIND=8) DIMENSION(*) for dfdintegrate1d/ dfdintegrateex1d	Array that contains additional information about the structure of partition *x* and left-side integration limits. For details on the *ldatahint* array, see table "Structure of the *datahint* Array" in the description of the df? intepolate1d function.

Name	Type	Description
rdatahint	REAL(KIND=4) DIMENSION(*) for dfsintegrate1d/ dfsintegrateex1d REAL(KIND=8) DIMENSION(*) for dfdintegrate1d/ dfdintegrateex1d	Array that contains additional information about the structure of partition *x* and right-side integration limits. For details on the *rdatahint* array, see table "Structure of the *datahint* Array" in the description of the df? intepolate1d function.
rhint	INTEGER	A flag describing the structure of the results. For the list of possible values of *rhint*, see table "Hint Values for Integration Results". If you set the flag to the DF_NO_HINT value, the library stores the results in row-major format.
le_cb	INTEGER	User-defined callback function for integration on interval [*llim*(*i*), min(*rlim*(*i*), *a*)) for *llim*(*i*) < *a* .
le_params	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the *le_cb* function.
re_cb	INTEGER	User-defined callback function for integration on interval [max(*llim*(*i*), *b*), *rlim*(*i*)) for *rlim*(*i*) ≥ *b*.
re_params	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the *re_cb* function.
i_cb	INTEGER	User-defined callback function for integration on interval [max(*a*, *llim*(*i*),), min(*rlim*(*i*), *b*)).
i_params	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the *i_cb* function.
search_cb	INTEGER	User-defined callback function for computing indices of cells that can contain interpolation sites.
search_params	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the *search_cb* function.

Output Parameters

Name	Type	Description
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Name	Type	Description
r	`REAL(KIND=4) DIMENSION(*)` for `df?integrate1d`/ `df?integrateex1d` `REAL(KIND=8) DIMENSION(*)` for `df?integrate1d`/ `df?integrateex1d`	Array of `integration` results. The size of the array should be sufficient to hold *nlim***ny* values, where *ny* is the dimension of the vector-valued function. The integration results are packed according to the settings in *rhint*.

Description

The `df?integrate1d`/`df?integrateex1d` routine computes spline-based integral on user-defined intervals

$$I(i,j)=\int_{ll_i}^{rl_i} f_j(x)dx$$

,

where rl_i = *rlim*(*i*), ll_i = *llim*(*i*), and *i* = 1, ..., *ny*.

If *rlim*(*i*) < *llim*(*i*), the routine returns

$$I(i,j)=-\int_{rl_i}^{ll_i} f_j(x)dx$$

The routine supports the following hint values for integration results:

Hint Values for Integration Results

Value	Description
`DF_MATRIX_STORAGE_ROWS`	Data is stored in row-major format according to C conventions.
`DF_MATRIX_STORAGE_COLS`	Data is stored in column-major format according to Fortran conventions.
`DF_NO_HINT`	No hint is provided. By default, the coordinates of vector-valued function *y* are provided and stored in row-major format.

A common structure of the storage formats for the integration results is as follows:

- Row-major format

I(1, 1)	...	*I*(1, *nlim*)
...	...	...
I (*ny*, 1)	...	*I*(*ny* , *nlim*)

- Column-major format

I(1, 1)	...	*I* (*ny*, 1)
...	...	...
I(1, *nlim*)	...	*I*(*ny* , *nlim*)

Using the *llimhint* and *rlimhint* parameters, you can provide the following hint values for integration limits:

Hint Values for Integration Limits

Value	Description
DF_SORTED_DATA	Integration limits are sorted in the ascending order and define a non-uniform partition.
DF_NON_UNIFORM_PARTITION	Partition defined by integration limits is non-uniform.
DF_UNIFORM_PARTITION	Partition defined by integration limits is uniform.
DF_NO_HINT	No hint is provided. By default, partition defined by integration limits is interpreted as non-uniform.

To compute integration with splines unsupported in the Data Fitting component, use the extended version of the routine df?integrateex1d. With this routine, you can provide user-defined callback functions that compute:

- integrals within, to the left of, or to the right of the interpolation interval [*a*, *b*]
- indices of cells that contain the provided integration limits or can serve as an approximation for computing the exact indices of such cells

If you do not pass callback functions, the routine uses the default settings.

See Also

Mathematical Conventions for Data Fitting Functions

df?interpolate1d/df?interpolateex1d

df?integrcallback

df?searchcellscallback

df?searchcells1d/df?searchcellsex1d

Searches sub-intervals containing interpolation sites.

Syntax

status = dfssearchcells1d(*task*, *method*, *nsite*, *site*, *sitehint*, *datahint*, *cell*)

status = dfdsearchcells1d(*task*, *method*, *nsite*, *site*, *sitehint*, *datahint*, *cell*)

status = dfssearchcellsex1d(*task*, *method*, *nsite*, *site*, *sitehint*, *datahint*, *cell*, *search_cb*, *search_params*)

status = dfdsearchcellsex1d(*task*, *method*, *nsite*, *site*, *sitehint*, *datahint*, *cell*, *search_cb*, *search_params*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task.
method	INTEGER	Search method. The supported value is DF_METHOD_STD.

Name	Type	Description
nsite	INTEGER	Number of interpolation sites.
site	REAL(KIND=4) DIMENSION(*) for dfssearchcells1d/ dfssearchcellsex1d REAL(KIND=8) DIMENSION(*) for dfdsearchcells1d/ dfdsearchcellsex1d	Array of interpolation sites of size *nsite*. The structure of the array is defined by the *sitehint* parameter: • If the sites form a non-uniform partition, the array should contain *nsite* values. • If the sites form a uniform partition, the array should contain two entries that represent the left-most and the right-most interpolation sites. The first entry of the array contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point.
sitehint	INTEGER	A flag describing the structure of the interpolation sites. For the list of possible values of *sitehint*, see table "Hint Values for Interpolation Sites". If you set the flag to DF_NO_HINT, the library interprets the site-defined partition as non-uniform.
datahint	REAL(KIND=4) DIMENSION(*) for dfssearchcells1d/ dfssearchcellsex1d REAL(KIND=8) DIMENSION(*) for dfdsearchcells1d/ dfdsearchcellsex1d	Array that contains additional information about the structure of the partition and interpolation sites. This data helps to speed up the computation. If you provide a NULL pointer, the routine uses the default settings for computations. For details on the *datahint* array, see table "Structure of the *datahint* Array".
search_cb	INTEGER	User-defined callback function for computing indices of cells that can contain interpolation sites. Set to NULL if you are not supplying a callback function.
search_params	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the *search_cb* function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.

Output Parameters

Name	Type	Description
status	INTEGER	Status of the routine: • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
cell	INTEGER DIMENSION(*)	Array of cell indices in the partition that contain the interpolation sites.

Description

The df?searchcells1d/df?searchcellsex1d routines return array *cell* of indices of sub-intervals (cells) in the partition that contain interpolation sites available in array *site*. For details on the cell indexing scheme, see the description of the df?interpolate1d/df?interpolateex1d computation routines.

Use the *datahint* parameter to provide additional information about the structure of the partition and/or interpolation sites. The definition of the *datahint* parameter is availalbe in the description of the df?interpolate1d/df?interpolateex1d computation routines.

For description of the user-defined callback for computation of cell indices, see df?searchcellscallback.

See Also

Mathematical Conventions for Data Fitting Functions

df?interpolate1d/df?interpolateex1d

df?searchcellscallback

df?interpcallback

A callback function for user-defined interpolation to be passed into df?interpolateex1d*.*

Syntax

status = dfsinterpcallback(*n*, *cell*, *site*, *r*, *user_params*, *library_params*)

status = dfdinterpcallback(*n*, *cell*, *site*, *r*, *user_params*, *library_params*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
n	INTEGER(KIND=8)	Number of interpolation sites.
cell	INTEGER(KIND=8) DIMENSION(*)	Array of size *n* containing indices of the cells to which the interpolation sites in array *site* belong.
site	REAL(KIND=4) DIMENSION(*) for dfsinterpcallback REAL(KIND=8) DIMENSION(*) for dfdinterpcallback	Array of interpolation sites of size *n*.
user_params	INTEGER DIMENSION(*), optional	Pointer to user-defined parameters of the callback function.
library_params	TYPE(DF_INTERP_CALLBACK_LIBRARY_PARAMS), optional	Pointer to library-defined parameters of the callback function.

Output Parameters

Name	Type	Description
status	INTEGER	The status returned by the callback function: • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • A positive value indicates a warning. See "Task Status and Error Reporting" for error code definitions.
r	REAL(KIND=4) DIMENSION(*) for dfsinterpcallback REAL(KIND=8) DIMENSION(*) for dfdinterpcallback	Array of the computed interpolation results packed in row-major format.

Description

When passed into the df?interpolateex1d routine, this function performs user-defined interpolation operation.

The *library_params* parameter allows the library to provide extra parameters, which the callback function can use to organize computations effectively. Currently no parameters are provided.

See Also

df?interpolate1d/df?interpolateex1d

df?searchcellscallback

df?integrcallback

A callback function that you can pass into df?integrateex1d *to define integration computations.*

Syntax

status = dfsintegrcallback(*n*, *lcell*, *llim*, *rcell*, *rlim*, *r*, *user_params*, *library_params*)

status = dfdintegrcallback(*n*, *lcell*, *llim*, *rcell*, *rlim*, *r*, *user_params*, *library_params*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
n	INTEGER(KIND=8)	Number of pairs of integration limits.
lcell	INTEGER(KIND=8) DIMENSION(*)	Array of size *n* with indices of the cells that contain the left-side integration limits in array *llim*.
llim	REAL(KIND=4) DIMENSION(*) for dfsintegrcallback	Array of size *n* that holds the left-side integration limits.

Name	Type	Description
	`REAL(KIND=8) DIMENSION(*)` for `dfdintegrcallback`	
rcell	`INTEGER(KIND=8) DIMENSION(*)`	Array of size *n* with indices of the cells that contain the right-side integration limits in array *rlim*.
rlim	`REAL(KIND=4) DIMENSION(*)` for `dfsintegrcallback` `REAL(KIND=8) DIMENSION(*)` for `dfdintegrcallback`	Array of size *n* that holds the right-side integration limits.
user_params	`INTEGER DIMENSION(*)`, optional	Pointer to user-defined parameters of the callback function.
library_params	`TYPE(DF_INTEGR_CALLBACK_LIBRARY_PARAMS)`, optional	Pointer to library-defined parameters of the callback function.

Output Parameters

Name	Type	Description
status	`INTEGER`	The status returned by the callback function: • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • A positive value indicates a warning. See "Task Status and Error Reporting" for error code definitions.
r	`REAL(KIND=4) DIMENSION(*)` for `dfsintegrcallback` `REAL(KIND=8) DIMENSION(*)` for `dfdintegrcallback`	Array of integration results. For packing the results in row-major format, follow the instructions described in `df?interpolate1d/df?interpolateex1d`.

Description

When passed into the `df?integrateex1d` routine, this function defines integration computations. If at least one of the integration limits is outside the interpolation interval [*a*, *b*], the library decomposes the integration into sub-intervals that belong to the extrapolation range to the left of *a*, the extrapolation range to the right of *b*, and the interpolation interval [*a*, *b*], as follows:

- If the left integration limit is to the left of the interpolation interval (*llim*< *a*), the `df?integrateex1d` routine passes *llim* as the left integration limit and `min`(*rlim*, *a*) as the right integration limit to the user-defined callback function.
- If the right integration limit is to the right of the interpolation interval (*rlim*> *b*), the `df?integrateex1d` routine passes `max`(*llim*, *b*) as the left integration limit and *rlim* as the right integration limit to the user-defined callback function.
- If the left and the right integration limits belong to the interpolation interval, the `df?integrateex1d` routine passes them to the user-defined callback function unchanged.

The value of the integral is the sum of integral values obtained on the sub-intervals.

The *library_params* parameter allows the library to provide extra parameters, which the callback function can use to organize computations effectively. Currently no parameters are provided.

See Also

df?integrate1d/df?integrateex1d

df?integrcallback

df?searchcellscallback

df?searchcellscallback

A callback function for user-defined search to be passed into df?interpolateex1d*,* df?integrateex1d*, or* df?searchcellsex1d*.*

Syntax

status = dfssearchcellscallback(*n*, *site*, *cell*, *flag*, *user_params*, *library_params*)

status = dfdsearchcellscallback(*n*, *site*, *cell*, *flag*, *user_params*, *library_params*)

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
n	INTEGER(KIND=8)	Number of interpolation sites or integration limits.
site	REAL(KIND=4) DIMENSION(*) for dfssearchcellscallback REAL(KIND=8) DIMENSION(*) for dfdsearchcellscallback	Array, size *n*, of interpolation sites or integration limits.
flag	INTEGER(KIND=4) DIMENSION(*)	Array of size *n*, with values set as follows: • If the cell with index *cell(i)* contains *site(i)*, set *flag*[*i*] to 1. • Otherwise, set *flag(i)* to zero. In this case, the library interprets the index as an approximation and computes the index of the cell containing *site(i)* by using the provided index as a starting point for the search.
user_params	INTEGER DIMENSION(*), optional	Pointer to user-defined parameters of the callback function.

Name	Type	Description
library_params	TYPE(DF_SEARCH_CALLBACK_LIBRARY_PARAMS), optional	Pointer to library-defined parameters of the callback function.

Output Parameters

Name	Type	Description
status	INTEGER	The status returned by the callback function: • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • The DF_STATUS_EXACT_RESULT status indicates that cell indices returned by the callback function are exact. In this case, you do not need to initialize entries of the *flag* array. • A positive value indicates a warning. See "Task Status and Error Reporting" for error code definitions.
cell	INTEGER(KIND=8) DIMENSION(*)	Array of size *n* that returns indices of the cells computed by the callback function.

Description

When passed into the df?interpolateex1d, df?integrateex1d, or df?searchcellsex1d routine, this function performs a user-defined search.

The *library_params* parameter allows the library to provide extra parameters, which the callback function can use to organize computations effectively. The df?interpolateex1d, and df?searchcellsex1d routines do not provide extra parameters. The df?integrateex1d routines use this parameter to specify which type of integration limits, left or right, are provided for the callback. To do this the library declares the DF_SEARCH_CALLBACK_LIBRARY_PARAMS derived type. It currently contains one component, limit_type_flag, of type INTEGER(KIND=4). The field is set by the library to one of two possible values: DF_INTEGR_SEARCH_CB_LLIM_FLAG if the left integration limits are provided, or DF_INTEGR_SEARCH_CB_RLIM_FLAG if the right integration limits are provided.

See Also

df?interpolate1d/df?interpolateex1d

df?interpcallback

Data Fitting Task Destructors

Task destructors are routines used to delete task descriptors and deallocate the corresponding memory resources. The Data Fitting task destructor dfdeletetask destroys a Data Fitting task and frees the memory.

dfdeletetask

Destroys a Data Fitting task object and frees the memory.

Syntax

```
status = dfdeletetask(task)
```

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
task	TYPE(DF_TASK)	Descriptor of the task to destroy.

Output Parameters

Name	Type	Description
status	INTEGER	Status of the routine: • DF_STATUS_OK if the task is deleted successfully. • Non-zero error code if the operation failed. See "Task Status and Error Reporting" for error code definitions.

Description

Given a pointer to a task descriptor, this routine deletes the Data Fitting task descriptor and frees the memory allocated for the structure. If the task is deleted successfully, the routine sets the task pointer to NULL. Otherwise, the routine returns an error code.

Appendix A: Linear Solvers Basics

Many applications in science and engineering require the solution of a system of linear equations. This problem is usually expressed mathematically by the matrix-vector equation, $Ax = b$, where A is an m-by-n matrix, x is the n element column vector and b is the m element column vector. The matrix A is usually referred to as the coefficient matrix, and the vectors x and b are referred to as the solution vector and the right-hand side, respectively.

Basic concepts related to solving linear systems with sparse matrices are described in Sparse Linear Systems and various storage schemes for sparse matrices are described in Sparse Matrix Storage Formats.

Sparse Linear Systems

In many real-life applications, most of the elements in A are zero. Such a matrix is referred to as sparse. Conversely, matrices with very few zero elements are called dense. For sparse matrices, computing the solution to the equation $Ax = b$ can be made much more efficient with respect to both storage and computation time, if the sparsity of the matrix can be exploited. The more an algorithm can exploit the sparsity without sacrificing the correctness, the better the algorithm.

Generally speaking, computer software that finds solutions to systems of linear equations is called a solver. A solver designed to work specifically on sparse systems of equations is called a sparse solver. Solvers are usually classified into two groups - direct and iterative.

Iterative Solvers start with an initial approximation to a solution and attempt to estimate the difference between the approximation and the true result. Based on the difference, an iterative solver calculates a new approximation that is closer to the true result than the initial approximation. This process is repeated until the difference between the approximation and the true result is sufficiently small. The main drawback to iterative solvers is that the rate of convergence depends greatly on the values in the matrix *A*. Consequently, it is not possible to predict how long it will take for an iterative solver to produce a solution. In fact, for ill-conditioned matrices, the iterative process will not converge to a solution at all. However, for well-conditioned matrices it is possible for iterative solvers to converge to a solution very quickly. Consequently, if an application involves well-conditioned matrices iterative solvers can be very efficient.

Direct Solvers, on the other hand, factor the matrix *A* into the product of two triangular matrices and then perform a forward and backward triangular solve.

This approach makes the time required to solve a systems of linear equations relatively predictable, based on the size of the matrix. In fact, for sparse matrices, the solution time can be predicted based on the number of non-zero elements in the array *A*.

Matrix Fundamentals

A matrix is a rectangular array of either real or complex numbers. A matrix is denoted by a capital letter; its elements are denoted by the same lower case letter with row/column subscripts. Thus, the value of the element in row *i* and column *j* in matrix *A* is denoted by `a(i,j)`. For example, a 3 by 4 matrix *A*, is written as follows:

$$A = \begin{bmatrix} a(1,1) & a(1,2) & a(1,3) & a(1,4) \\ a(2,1) & a(2,2) & a(2,3) & a(2,4) \\ a(3,1) & a(3,2) & a(3,3) & a(3,4) \end{bmatrix}$$

Note that with the above notation, we assume the standard Fortran programming language convention of starting array indices at 1 rather than the C programming language convention of starting them at 0.

A matrix in which all of the elements are real numbers is called a real matrix. A matrix that contains at least one complex number is called a complex matrix. A real or complex matrix *A* with the property that *a*(*i*,*j*) = *a*(*j*,*i*), is called a symmetric matrix. A complex matrix *A* with the property that *a*(*i*,*j*) = *conj*(*a*(*j*,*i*)), is called a Hermitian matrix. Note that programs that manipulate symmetric and Hermitian matrices need only store half of the matrix values, since the values of the non-stored elements can be quickly reconstructed from the stored values.

A matrix that has the same number of rows as it has columns is referred to as a square matrix. The elements in a square matrix that have same row index and column index are called the diagonal elements of the matrix, or simply the diagonal of the matrix.

The transpose of a matrix *A* is the matrix obtained by "flipping" the elements of the array about its diagonal. That is, we exchange the elements `a(i,j)` and `a(j,i)`. For a complex matrix, if we both flip the elements about the diagonal and then take the complex conjugate of the element, the resulting matrix is called the Hermitian transpose or conjugate transpose of the original matrix. The transpose and Hermitian transpose of a matrix *A* are denoted by A^T and A^H respectively.

A column vector, or simply a vector, is a $n \times 1$ matrix, and a row vector is a $1 \times n$ matrix. A real or complex matrix *A* is said to be positive definite if the vector-matrix product x^TAx is greater than zero for all non-zero vectors *x*. A matrix that is not positive definite is referred to as indefinite.

An upper (or lower) triangular matrix, is a square matrix in which all elements below (or above) the diagonal are zero. A unit triangular matrix is an upper or lower triangular matrix with all 1's along the diagonal.

A matrix P is called a permutation matrix if, for any matrix A, the result of the matrix product PA is identical to A except for interchanging the rows of A. For a square matrix, it can be shown that if PA is a permutation of the rows of A, then AP^T is the same permutation of the columns of A. Additionally, it can be shown that the inverse of P is P^T.

In order to save space, a permutation matrix is usually stored as a linear array, called a permutation vector, rather than as an array. Specifically, if the permutation matrix maps the i-th row of a matrix to the j-th row, then the i-th element of the permutation vector is j.

A matrix with non-zero elements only on the diagonal is called a diagonal matrix. As is the case with a permutation matrix, it is usually stored as a vector of values, rather than as a matrix.

Direct Method

For solvers that use the direct method, the basic technique employed in finding the solution of the system $Ax = b$ is to first factor A into triangular matrices. That is, find a lower triangular matrix L and an upper triangular matrix U, such that $A = LU$. Having obtained such a factorization (usually referred to as an LU decomposition or LU factorization), the solution to the original problem can be rewritten as follows.

$$Ax = b$$
$$\Rightarrow \quad LUx = b$$
$$\Rightarrow \quad L(Ux) = b$$

This leads to the following two-step process for finding the solution to the original system of equations:

1. Solve the systems of equations $Ly = b$.
2. Solve the system $Ux = y$.

Solving the systems $Ly = b$ and $Ux = y$ is referred to as a forward solve and a backward solve, respectively.

If a symmetric matrix A is also positive definite, it can be shown that A can be factored as LL^T where L is a lower triangular matrix. Similarly, a Hermitian matrix, A, that is positive definite can be factored as $A = LL^H$. For both symmetric and Hermitian matrices, a factorization of this form is called a Cholesky factorization.

In a Cholesky factorization, the matrix U in an LU decomposition is either L^T or L^H. Consequently, a solver can increase its efficiency by only storing L, and one-half of A, and not computing U. Therefore, users who can express their application as the solution of a system of positive definite equations will gain a significant performance improvement over using a general representation.

For matrices that are symmetric (or Hermitian) but not positive definite, there are still some significant efficiencies to be had. It can be shown that if A is symmetric but not positive definite, then A can be factored as $A = LDL^T$, where D is a diagonal matrix and L is a lower unit triangular matrix. Similarly, if A is Hermitian, it can be factored as $A = LDL^H$. In either case, we again only need to store L, D, and half of A and we need not compute U. However, the backward solve phases must be amended to solving $L^Tx = D^{-1}y$ rather than $L^Tx = y$.

Fill-In and Reordering of Sparse Matrices

Two important concepts associated with the solution of sparse systems of equations are fill-in and reordering. The following example illustrates these concepts.

Consider the system of linear equation $Ax = b$, where A is a symmetric positive definite sparse matrix, and A and b are defined by the following:

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ \frac{3}{2} & \frac{1}{2} & \star & \star & \star \\ 6 & \star & 12 & \star & \star \\ \frac{3}{4} & \star & \star & \frac{5}{8} & \star \\ 3 & \star & \star & \star & 16 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

A star (*) is used to represent zeros and to emphasize the sparsity of *A*. The Cholesky factorization of *A* is: $A = LL^T$, where *L* is the following:

$$L = \begin{bmatrix} 3 & * & * & * & * \\ \frac{1}{2} & \frac{1}{2} & * & * & * \\ 2 & -2 & 2 & * & * \\ \frac{1}{4} & \frac{1}{-4} & \frac{1}{-2} & \frac{1}{2} & * \\ 1 & -1 & -2 & -3 & 1 \end{bmatrix}$$

Notice that even though the matrix *A* is relatively sparse, the lower triangular matrix *L* has no zeros below the diagonal. If we computed *L* and then used it for the forward and backward solve phase, we would do as much computation as if *A* had been dense.

The situation of *L* having non-zeros in places where *A* has zeros is referred to as fill-in. Computationally, it would be more efficient if a solver could exploit the non-zero structure of *A* in such a way as to reduce the fill-in when computing *L*. By doing this, the solver would only need to compute the non-zero entries in *L*. Toward this end, consider permuting the rows and columns of *A*. As described in Matrix Fundamentals, the permutations of the rows of *A* can be represented as a permutation matrix, *P*. The result of permuting the rows is the product of *P* and *A*. Suppose, in the above example, we swap the first and fifth row of *A*, then swap the first and fifth columns of *A*, and call the resulting matrix *B*. Mathematically, we can express the process of permuting the rows and columns of *A* to get *B* as $B = PAP^T$. After permuting the rows and columns of *A*, we see that *B* is given by the following:

$$B = \begin{bmatrix} 16 & * & * & * & 3 \\ * & \frac{1}{2} & * & * & \frac{3}{2} \\ * & * & 12 & * & 6 \\ * & * & * & \frac{5}{8} & \frac{3}{4} \\ 3 & \frac{3}{2} & 6 & \frac{3}{4} & 9 \end{bmatrix}$$

Since *B* is obtained from *A* by simply switching rows and columns, the numbers of non-zero entries in *A* and *B* are the same. However, when we find the Cholesky factorization, $B = LL^T$, we see the following:

$$L = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix}$$

The fill-in associated with *B* is much smaller than the fill-in associated with *A*. Consequently, the storage and computation time needed to factor *B* is much smaller than to factor *A*. Based on this, we see that an efficient sparse solver needs to find permutation *P* of the matrix *A*, which minimizes the fill-in for factoring $B = PAP^T$, and then use the factorization of *B* to solve the original system of equations.

Although the above example is based on a symmetric positive definite matrix and a Cholesky decomposition, the same approach works for a general *LU* decomposition. Specifically, let *P* be a permutation matrix, $B = PAP^T$ and suppose that *B* can be factored as $B = LU$. Then

$$Ax = b$$

$$\Rightarrow$$

$$PA(P^{-1}P)x = Pb$$

$$\Rightarrow$$

$$PA(P^TP)x = Pb$$

$$\Rightarrow$$

$$(PAP^T)(Px) = Pb$$

$$\Rightarrow$$

$$B(Px) = Pb$$

$$\Rightarrow$$

$$LU(Px) = Pb$$

It follows that if we obtain an *LU* factorization for *B*, we can solve the original system of equations by a three step process:

1. Solve $Ly = Pb$.
2. Solve $Uz = y$.
3. Set $x = P^T z$.

If we apply this three-step process to the current example, we first need to perform the forward solve of the systems of equation $Ly = Pb$:

$$Ly = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix} * \begin{bmatrix} y1 \\ y2 \\ y3 \\ y4 \\ y5 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 4 \\ 1 \end{bmatrix}$$

This gives:

$$y^T = \frac{5}{4}, 2\sqrt{2}, \frac{\sqrt{3}}{2}, \frac{16}{\sqrt{10}}, \frac{-979\sqrt{\frac{3}{5}}}{12}.$$

The second step is to perform the backward solve, $Uz = y$. Or, in this case, since a Cholesky factorization is used, $L^T z = y$.

$$\begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix}^T * \begin{bmatrix} z1 \\ z2 \\ z3 \\ z4 \\ z5 \end{bmatrix} = \begin{bmatrix} \frac{5}{4} \\ 2(\sqrt{2}) \\ \frac{\sqrt{3}}{2} \\ \frac{16}{\sqrt{10}} \\ \frac{-979\sqrt{\frac{3}{5}}}{12} \end{bmatrix}$$

This gives

$$z^T = \frac{123}{2}, 983, \frac{1961}{12}, 398, \frac{-979}{3}.$$

The third and final step is to set $x = P^T z$. This gives

$$x^T = \frac{-979}{3}, 983, \frac{1961}{12}, 398, \frac{123}{2}.$$

Sparse Matrix Storage Formats

It is more efficient to store only the non-zero elements of a sparse matrix. There are a number of common storage formats used for sparse matrices, but most of them employ the same basic technique. That is, store all non-zero elements of the matrix into a linear array and provide auxiliary arrays to describe the locations of the non-zero elements in the original matrix.

Storage Formats for the Direct Sparse Solvers

Storing the non-zero elements of a sparse matrix into a linear array is done by walking down each column (column-major format) or across each row (row-major format) in order, and writing the non-zero elements to a linear array in the order they appear in the walk.

- DSS Symmetric Matrix Storage
- DSS Nonsymmetric Matrix Storage
- DSS Structurally Symmetric Matrix Storage
- DSS Distributed Symmetric Matrix Storage

Sparse Matrix Storage Formats for Sparse BLAS Levels 2 and Level 3

These sections describe in detail the sparse matrix storage formats supported in the current version of the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3.

- Sparse BLAS CSR Matrix Storage
- Sparse BLAS CSC Matrix Storage
- Sparse BLAS Coordinate Matrix Storage
- Sparse BLAS Diagonal Matrix Storage
- Sparse BLAS Skyline Matrix Storage
- Sparse BLAS BSR Matrix Storage

DSS Symmetric Matrix Storage

For symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular format) or the lower triangular half of the matrix (lower triangular format).

The Intel® oneAPI Math Kernel Library (oneMKL) direct sparse solvers use a row-major upper triangular storage format: the matrix is compressed row-by-row and for symmetric matrices only non-zero elements in the upper triangular half of the matrix are stored.

The Intel® oneAPI Math Kernel Library (oneMKL) sparse matrix storage format for direct sparse solvers is specified by three arrays:*values*, *columns*, and *rowIndex*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix.

values	A real or complex array that contains the non-zero elements of a sparse matrix. The non-zero elements are mapped into the *values* array using the row-major upper triangular storage mapping described above.
columns	Element *i* of the integer array *columns* is the number of the column that contains the *i*-th element in the *values* array.
rowIndex	Element *j* of the integer array *rowIndex* gives the index of the element in the *values* array that is first non-zero element in a row *j*.

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in the matrix.

As the *rowIndex* array gives the location of the first non-zero element within a row, and the non-zero elements are stored consecutively, the number of non-zero elements in the *i*-th row is equal to the difference of `rowIndex(i)` and `rowIndex(i+1)`.

To have this relationship hold for the last row of the matrix, an additional entry (dummy entry) is added to the end of *rowIndex*. Its value is equal to the number of non-zero elements plus one. This makes the total length of the *rowIndex* array one larger than the number of rows in the matrix.

NOTE
The Intel® oneAPI Math Kernel Library (oneMKL) sparse storage scheme for the direct sparse solvers supports both one-based indexing and zero-based indexing.

Consider the symmetric matrix *A*:

$$A = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -3 & * & 6 & 7 & * \\ * & * & 4 & * & -5 \end{pmatrix}$$

Only elements from the upper triangle are stored. The actual arrays for the matrix *A* are as follows:

Storage Arrays for a Symmetric Matrix

one-based indexing									
values	=	(1	-1	-3	5	4	6	4	7	-5)
columns	=	(1	2	4	2	3	4	5	4	5)
rowIndex	=	(1	4	5	8	9	10)			
zero-based indexing										

values	=	(1	-1	-3	5	4	6	4	7	-5)
columns	=	(0	1	3	1	2	3	4	3	4)
rowIndex	=	(0	3	4	7	8	9)			

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

DSS Nonsymmetric Matrix Storage

For a non-symmetric or non-Hermitian matrix, all non-zero elements need to be stored. Consider the non-symmetric matrix *B*:

$$\begin{pmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{pmatrix}$$

The matrix *B* has 13 non-zero elements, and all of them are stored as follows:

Storage Arrays for a Non-Symmetric Matrix

one-based indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
rowIndex	=	(1	4	6	9	12	14)							
zero-based indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
rowIndex	=	(0	3	5	8	11	13)							

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

DSS Structurally Symmetric Matrix Storage

Direct sparse solvers can also solve symmetrically structured systems of equations. A symmetrically structured system of equations is one where the pattern of non-zero elements is symmetric. That is, a matrix has a symmetric structure if $a_{j,i}$ is not zero if and only if $a_{i,j}$ is not zero. From the point of view of the solver software, a "non-zero" element of a matrix is any element stored in the *values* array, even if its value is

equal to 0. In that sense, any non-symmetric matrix can be turned into a symmetrically structured matrix by carefully adding zeros to the *values* array. For example, the above matrix *B* can be turned into a symmetrically structured matrix by adding two non-zero entries:

$$B = \begin{pmatrix} 1 & -1 & * & 3 & * \\ -2 & 5 & * & * & 0 \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & 0 & * & -5 \end{pmatrix}$$

The matrix *B* can be considered to be symmetrically structured with 15 non-zero elements and represented as:

Storage Arrays for a Symmetrically Structured Matrix

one-based indexing															
values	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)
columns	=	(1	2	4	1	2	5	3	4	5	1	3	4	2	3	5)
rowIndex	=	(1	4	7	10	13	16)									
zero-based indexing																
values	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)
columns	=	(0	1	3	0	1	4	2	3	4	0	2	3	1	2	4)
rowIndex	=	(0	3	6	9	12	15)									

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

DSS Distributed Symmetric Matrix Storage

The distributed assembled matrix input format can be used by the Parallel Direct Sparse Solver for Clusters Interface.

In this format, the symmetric input matrix *A* is divided into sequential row subsets, or domains. Each domain belongs to an MPI process. Neighboring domains can overlap. For such intersection between two domains, the element values of the full matrix can be obtained by summing the respective elements of both domains.

As in the centralized format, the distributed format uses three arrays to describe the input data, but the *values*, *columns*, and *rowIndex* arrays on each processor only describe the domain belonging to that particular processor and not the entire matrix.

For example, consider a symmetric matrix *A*:

$$A = \begin{pmatrix} \mathbf{6} & \mathbf{-1} & * & \mathbf{-3} & * \\ \mathbf{-1} & \mathbf{5} & * & * & * \\ * & * & \mathbf{11} & \mathbf{5} & \mathbf{4} \\ -3 & * & 5 & \mathbf{10} & * \\ * & * & 4 & * & \mathbf{5} \end{pmatrix}$$

This array could be distributed between two domains corresponding to two MPI processes, with the first containing rows 1 through 3, and the second containing rows 3 through 5.

NOTE
For the symmetric input matrix, it is not necessary to store the values from the lower triangle.

$$A_{Domain1} = \begin{pmatrix} \mathbf{6} & \mathbf{-1} & * & \mathbf{-3} & * \\ \mathbf{-1} & \mathbf{5} & * & * & * \\ * & * & \mathbf{3} & * & \mathbf{2} \end{pmatrix}$$

Distributed Storage Arrays for a Symmetric Matrix, Domain 1

one-based indexing							
values	=	(6	-1	-3	5	3	2)
columns	=	(1	2	4	2	3	5)
rowIndex	=	(1	4	5	7)		
zero-based indexing							
values	=	(6	-1	-3	5	3	2)
columns	=	(0	1	3	1	2	4)
rowIndex	=	(0	3	4	6)		

$$A_{Domain2} = \begin{pmatrix} * & * & \mathbf{8} & \mathbf{5} & \mathbf{2} \\ \mathbf{-3} & * & \mathbf{5} & \mathbf{10} & * \\ * & * & \mathbf{4} & * & \mathbf{5} \end{pmatrix}$$

Distributed Storage Arrays for a Symmetric Matrix, Domain 2

one-based indexing						
values	=	(8	5	2	10	5)
columns	=	(3	4	5	4	5)
rowIndex	=	(1	4	5	6)	
zero-based indexing						
values	=	(8	5	2	10	5)
columns	=	(2	3	4	3	4)
rowIndex	=	(0	3	4	5)	

The third row of matrix *A* is common between domain 1 and domain 2. The values of row 3 of matrix *A* are the sums of the respective elements of row 3 of matrix $A_{Domain1}$ and row 1 of matrix $A_{Domain2}$.

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

Sparse BLAS CSR Matrix Storage Format

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS compressed sparse row (CSR) format is specified by four arrays:

- *values*

- *columns*
- *pointerB*
- *pointerE*

In addition, each sparse matrix has an associated variable ,*indexing*, which specifies if the matrix indices are 0-based (*indexing*=0) or 1-based (*indexing*=1). These are descriptions of the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

values	A real or complex array that contains the non-zero elements of *A*. Values of the non-zero elements of *A* are mapped into the *values* array using the row-major storage mapping described above.
columns	Element *i* of the integer array *columns* is the number of the column in *A* that contains the *i*-th value in the *values* array.
pointerB	Element *j* of this integer array gives the index of the element in the *values* array that is first non-zero element in a row *j* of *A*. Note that this index is equal to *pointerB(j) - indexing* .
pointerE	An integer array that contains row indices, such that *pointerE(j)-indexing* is the index of the element in the *values* array that is last non-zero element in a row *j* of *A*.

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in *A*.The length of the *pointerB* and *pointerE* arrays is equal to the number of rows in *A*.

NOTE
Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines support the CSR format both with one-based indexing and zero-based indexing.

You can represent the matrix *B*

$$B = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{pmatrix}$$

in the CSR format as:

Storage Arrays for a Matrix in CSR Format

one-based indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
pointerB	=	(1	4	6	9	12)								
pointerE	=	(4	6	9	12	14)								
zero-based indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
pointerB	=	(0	3	5	8	11)								
pointerE	=	(3	5	8	11	13)								

Additionally, you can define submatrices with different *pointerB* and *pointerE* arrays that share the same *values* and *columns* arrays of a CSR matrix. For example, you can represent the lower right 3x3 submatrix of *B* as:

Storage Arrays for a Matrix in CSR Format

one-based indexing				
subpointerB	=	(6	10	13)
subpointerE	=	(9	12	14)
zero-based indexing				

subpointerB	=	(5	9	12)
subpointerE	=	(8	11	13)

NOTE The CSR matrix must have a monotonically increasing row index. That is, *pointerB[i]* ≤ *pointerB[j]* and *pointerE[i]* ≤ *pointerE[j]* for all indices *i* <*j*.

This storage format is used in the NIST Sparse BLAS library [Rem05].

Three Array Variation of CSR Format

The storage format accepted for the direct sparse solvers is a variation of the CSR format. It also is used in the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 both with one-based indexing and zero-based indexing. The above matrix*B* can be represented in this format (referred to as the 3-array variation of the CSR format or CSR3) as:

Storage Arrays for a Matrix in CSR Format (3-Array Variation)

one-based indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
rowIndex	=	(1	4	6	9	12	14)							
zero-based indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
rowIndex	=	(0	3	5	8	11	13)							

The 3-array variation of the CSR format has a restriction: all non-zero elements are stored continuously, that is the set of non-zero elements in the row *J* goes just after the set of non-zero elements in the row *J–1*.

There are no such restrictions in the general (NIST) CSR format. This may be useful, for example, if there is a need to operate with different submatrices of the matrix at the same time. In this case, it is enough to define the arrays *pointerB* and *pointerE* for each needed submatrix so that all these arrays are pointers to the same array *values*.

By definition, the array *rowIndex* from the Table "Storage Arrays for a Non-Symmetric Example Matrix" is related to the arrays *pointerB* and *pointerE* from the Table "Storage Arrays for an Example Matrix in CSR Format", and you can see that

```
pointerB(i) = rowIndex(i) for i=1, ..5;
              pointerE(i) = rowIndex(i+1) for i=1, ..5.
```

This enables calling a routine that has *values*, *columns*, *pointerB* and *pointerE* as input parameters for a sparse matrix stored in the format accepted for the direct sparse solvers. For example, a routine with the interface:

```
   Subroutine name_routine(.... ,  values, columns, pointerB, pointerE, ...)
```

can be called with parameters *values*, *columns*, *rowIndex* as follows:

```
   call name_routine(.... ,  values, columns, rowIndex, rowIndex(2), ...).
```

Sparse BLAS CSC Matrix Storage Format

The compressed sparse column format (CSC) is similar to the CSR format, but the columns are used instead the rows. In other words, the CSC format is identical to the CSR format for the transposed matrix. The CSR format is specified by four arrays: *values*, *columns*, *pointerB*, and *pointerE*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

values A real or complex array that contains the non-zero elements of *A*. Values of the non-zero elements of *A* are mapped into the *values* array using the column-major storage mapping.

rows Element *i* of the integer array *rows* is the number of the row in *A* that contains the *i*-th value in the *values* array.

pointerB Element *j* of this integer array gives the index of the element in the *values* array that is first non-zero element in a column *j* of *A*. Note that this index is equal to `pointerB(j)-indexing` for Inspector-executor Sparse BLAS CSC arrays.

pointerE An integer array that contains column indices, such that `pointerE(j)-indexing` is the index of the element in the *values* array that is last non-zero element in a column *j* of *A*.

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in *A*. The length of the *pointerB* and *pointerE* arrays is equal to the number of columns in *A*.

NOTE
Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines support the CSC format both with one-based indexing and zero-based indexing.

For example, consider matrix *B*:

$$B = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{pmatrix}$$

It can be represented in the CSC format as:

Storage Arrays for a Matrix in CSC Format

one-based indexing													
values	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
rows	=	(1	2	4	1	2	5	3	4	1	3	4	3	5)
pointerB	=	(1	4	7	9	12)								
pointerE	=	(4	7	9	12	14)								
zero-based indexing													
values	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
rows	=	(0	1	3	0	1	4	2	3	0	2	3	2	4)
pointerB	=	(0	3	6	8	11)								
pointerE	=	(3	6	8	11	13)								

Sparse BLAS Coordinate Matrix Storage Format

The coordinate format is the most flexible and simplest format for the sparse matrix representation. Only non-zero elements are stored, and the coordinates of each non-zero element are given explicitly. Many commercial libraries support the matrix-vector multiplication for the sparse matrices in the coordinate format.

The Intel® oneAPI Math Kernel Library (oneMKL) coordinate format is specified by three arrays:*values*, *rows*, and *column*, and a parameter *nnz* which is number of non-zero elements in *A*. All three arrays have dimension *nnz*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

values A real or complex array that contains the non-zero elements of *A* in any order.

rows Element *i* of the integer array *rows* is the number of the row in *A* that contains the *i*-th value in the *values* array.

columns Element *i* of the integer array *columns* is the number of the column in *A* that contains the *i*-th value in the *values* array.

NOTE
Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines support the coordinate format both with one-based indexing and zero-based indexing.

For example, the sparse matrix *C*

$$C = \begin{pmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{pmatrix}$$

can be represented in the coordinate format as follows:

Storage Arrays for an Example Matrix in case of the coordinate format

one-based indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
rows	=	(1	1	1	2	2	3	3	3	4	4	4	5	5)
columns	=	(1	2	3	1	2	3	4	5	1	3	4	2	5)
zero-based indexing														
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
rows	=	(0	0	0	1	1	2	2	2	3	3	3	4	4)
columns	=	(0	1	2	0	1	2	3	4	0	2	3	1	4)

Sparse BLAS Diagonal Matrix Storage Format

If the sparse matrix has diagonals containing only zero elements, then the diagonal storage format can be used to reduce the amount of information needed to locate the non-zero elements. This storage format is particularly useful in many applications where the matrix arises from a finite element or finite difference discretization. The Intel® oneAPI Math Kernel Library (oneMKL) diagonal storage format is specified by two arrays:*values* and *distance*, and two parameters: *ndiag*, which is the number of non-empty diagonals, and *lval*, which is the declared leading dimension in the calling (sub)programs. The following table describes the arrays *values* and *distance*:

values A real or complex two-dimensional array is dimensioned as *lval* by *ndiag*. Each column of it contains the non-zero elements of certain diagonal of *A*. The key point of the storage is that each element in *values* retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom. Note that the value of *distance*(*i*) is the number of elements to be padded for diagonal *i*.

distance An integer array with dimension *ndiag*. Element *i* of the array *distance* is the distance between *i*-diagonal and the main diagonal. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

The above matrix *C* can be represented in the diagonal storage format as follows:

$$distance = (-3\ -1\ 0\ 1\ 2)$$

$$values = \begin{pmatrix} * & * & 1 & -1 & -3 \\ * & -2 & 5 & 0 & 0 \\ * & 0 & 4 & 6 & 4 \\ -4 & 2 & 7 & 0 & * \\ 8 & 0 & -5 & * & * \end{pmatrix}$$

where the asterisks denote padded elements.

When storing symmetric, Hermitian, or skew-symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For the Intel® oneAPI Math Kernel Library (oneMKL) triangular solver routines elements of the array*distance* must be sorted in increasing order. In all other cases the diagonals and distances can be stored in arbitrary order.

Sparse BLAS Skyline Matrix Storage Format

The skyline storage format is important for the direct sparse solvers, and it is well suited for Cholesky or LU decomposition when no pivoting is required.

The skyline storage format accepted in Intel® oneAPI Math Kernel Library (oneMKL) can store only triangular matrix or triangular part of a matrix. This format is specified by two arrays:*values* and *pointers*. The following table describes these arrays:

values	A scalar array. For a lower triangular matrix it contains the set of elements from each row of the matrix starting from the first non-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets.
pointers	An integer array with dimension `(m+1)`, where *m* is the number of rows for lower triangle (columns for the upper triangle). `pointers(i) - pointers(1)+1` gives the index of element in *values* that is first non-zero element in row (column) *i*. The value of `pointers(m+1)` is set to `nnz+pointers(1)`, where *nnz* is the number of elements in the array *values*.

For example, consider the matrix *C*:

$$C = \begin{pmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{pmatrix}$$

The low triangle of the matrix *C* given above can be stored as follows:

```
values  =  ( 1  -2   5   4  -4   0   2   7   8   0   0   -5 )
                pointers = ( 1   2   4   5   9   13 )
```

and the upper triangle of this matrix *C* can be stored as follows:

```
values   = ( 1  -1   5  -3   0   4   6   7  4   0   -5 )
                pointers = ( 1   2   4   7   9   12 )
```

This storage format is supported by the NIST Sparse BLAS library [Rem05].

Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines operating with the skyline storage format do not support general matrices.

Sparse BLAS BSR Matrix Storage Format

The Intel® oneAPI Math Kernel Library (oneMKL) block compressed sparse row (BSR) format for sparse matrices is specified by four arrays:*values*, *columns*, *pointerB*, and *pointerE*. The following table describes these arrays.

values	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block-by-block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them are equal to zero. Within each non-zero block elements are stored in column-major order in the case of one-based indexing, and in row-major order in the case of the zero-based indexing.
columns	Element *i* of the integer array *columns* is the number of the column in the block matrix that contains the *i*-th non-zero block.
pointerB	Element *j* of this integer array gives the index of the element in the *columns* array that is first non-zero block in a row *j* of the block matrix.
pointerE	Element *j* of this integer array gives the index of the element in the *columns* array that contains the last non-zero block in a row *j* of the block matrix plus 1.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks. The length of the *pointerB* and *pointerE* arrays is equal to the number of block rows in the block matrix.

NOTE
Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines support BSR format both with one-based indexing and zero-based indexing.

For example, consider the sparse matrix *D*

$$D = \begin{pmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ * & * & 1 & 4 & * & * \\ * & * & 5 & 1 & * & * \\ * & * & 4 & 3 & 7 & 2 \\ * & * & 0 & 0 & 0 & 0 \end{pmatrix}$$

If the size of the block equals 2, then the sparse matrix *D* can be represented as a 3x3 block matrix *E* with the following structure:

$$E = \begin{pmatrix} L & M & * \\ * & N & * \\ * & P & Q \end{pmatrix}$$

where

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, M = \begin{pmatrix} 6 & 7 \\ 8 & 2 \end{pmatrix}, N = \begin{pmatrix} 1 & 4 \\ 5 & 1 \end{pmatrix}, P = \begin{pmatrix} 4 & 3 \\ 0 & 0 \end{pmatrix}, Q = \begin{pmatrix} 7 & 2 \\ 0 & 0 \end{pmatrix}$$

The matrix *D* can be represented in the BSR format as follows:

one-based indexing

```
values  =  (1 2 0 1 6 8 7 2 1 5 4 1 4 0 3 0 7 0 2 0)
           columns  = (1   2   2   2   3)
           pointerB = (1   3   4)
           pointerE = (3   4   6)
```

zero-based indexing

```
values  =  [1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0]
           columns  = [0   1   1   1   2]
           pointerB = [0   2   3]
           pointerE = [2   3   5]
```

This storage format is supported by the NIST Sparse BLAS library [Rem05].

Three Array Variation of BSR Format

Intel® oneAPI Math Kernel Library (oneMKL) supports the variation of the BSR format that is specified by three arrays:*values*, *columns*, and *rowIndex*. The following table describes these arrays.

values	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block by block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Within each non-zero block the elements are stored in column major order in the case of the one-based indexing, and in row major order in the case of the zero-based indexing.
columns	Element *i* of the integer array *columns* is the number of the column in the block matrix that contains the *i*-th non-zero block.
rowIndex	Element *j* of this integer array gives the index of the element in the *columns* array that is first non-zero block in a row *j* of the block matrix.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks.

As the *rowIndex* array gives the location of the first non-zero block within a row, and the non-zero blocks are stored consecutively, the number of non-zero blocks in the *i*-th row is equal to the difference of `rowIndex(i)` and `rowIndex(i+1)`.

To retain this relationship for the last row of the block matrix, an additional entry (dummy entry) is added to the end of *rowIndex* with value equal to the number of non-zero blocks plus one. This makes the total length of the *rowIndex* array one larger than the number of rows of the block matrix.

The above matrix *D* can be represented in this 3-array variation of the BSR format as follows:

one-based indexing

```
values  =  (1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0)
           columns  = (1   2   2   2   3)
           rowIndex = (1   3   4   6)
```

zero-based indexing

```
values  =  (1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0)
           columns  = (0   1   1   1   2)
           rowIndex = (0   2   3 5)
```

When storing symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For example, consider the symmetric sparse matrix *F*:

$$F = \begin{pmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ 6 & 8 & 1 & 4 & * & * \\ 7 & 2 & 5 & 2 & * & * \\ * & * & * & * & 7 & 2 \\ * & * & * & * & 0 & 0 \end{pmatrix}$$

If the size of the block equals 2, then the sparse matrix *F* can be represented as a 3x3 block matrix *G* with the following structure:

$$G = \begin{pmatrix} L & M & * \\ M' & N & * \\ * & * & Q \end{pmatrix}$$

where

$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}$, $M = \begin{pmatrix} 6 & 7 \\ 8 & 2 \end{pmatrix}$, $M' = \begin{pmatrix} 6 & 8 \\ 7 & 2 \end{pmatrix}$, $N = \begin{pmatrix} 1 & 4 \\ 5 & 2 \end{pmatrix}$, and $Q = \begin{pmatrix} 7 & 2 \\ 0 & 0 \end{pmatrix}$

The symmetric matrix *F* can be represented in this 3-array variation of the BSR format (storing only the upper triangular part) as follows:

one-based indexing

```
values  =  (1 2 0 1 6 8 7 2 1 5 4 2 7 0 2 0)
               columns  = (1   2   2   3)
               rowIndex = (1   3   4 5)
```

zero-based indexing

```
values  =  (1 0 2 1 6 7 8 2 1 4 5 2 7 2 0 0)
               columns  = (0   1   1   2)
               rowIndex = (0   2   3 4)
```

Variable BSR Format

A variation of BSR3 is variable block compressed sparse row format. For a trust level *t*, $0 \le t \le 100$, rows similar up to *t* percent are placed in one supernode.

Appendix B: Routine and Function Arguments

The major arguments in the BLAS routines are vector and matrix, whereas VM functions work on vector arguments only. The sections that follow discuss each of these arguments and provide examples.

Vector Arguments in BLAS

Vector arguments are passed in one-dimensional arrays. The array dimension (length) and vector increment are passed as integer variables. The length determines the number of elements in the vector. The increment (also called stride) determines the spacing between vector elements and the order of the elements in the array in which the vector is passed.

A vector of length *n* and increment *incx* is passed in a one-dimensional array *x* whose values are defined as

```
x(1), x(1+|incx|), ..., x(1+(n-1)* |incx|)
```

If *incx* is positive, then the elements in array *x* are stored in increasing order. If *incx* is negative, the elements in array *x* are stored in decreasing order with the first element defined as `x(1+(n-1)* |incx|)`. If *incx* is zero, then all elements of the vector have the same value, `x(1)`. The size of the one-dimensional array that stores the vector must always be at least

```
idimx = 1 + (n-1)* |incx |
```

Example. One-dimensional Real Array

Let `x(1:7)` be the one-dimensional real array

```
x = (1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0).
```

If `incx =2` and `n = 3`, then the vector argument with elements in order from first to last is `(1.0, 5.0, 9.0)`.

If `incx = -2` and `n = 4`, then the vector elements in order from first to last is `(13.0, 9.0, 5.0, 1.0)`.

If `incx = 0` and `n = 4`, then the vector elements in order from first to last is `(1.0, 1.0, 1.0, 1.0)`.

One-dimensional substructures of a matrix, such as the rows, columns, and diagonals, can be passed as vector arguments with the starting address and increment specified.

In Fortran, storing the `m-by-n` matrix is based on column-major ordering where the increment between elements in the same column is `1`, the increment between elements in the same row is *m*, and the increment between elements on the same diagonal is `m + 1`.

Example. Two-dimensional Real Matrix

Let *a* be a real 5 x 4 matrix declared as `REAL A (5,4)float a[5*4];`.

To scale the third column of *a* by 2.0, use the BLAS routine `sscal` with the following calling sequence:

```
call sscal (5, 2.0, a(1,3), 1)
```

To scale the second row, use the statement:

```
call sscal (4, 2.0, a(2,1), 5)
```

To scale the main diagonal of *a* by 2.0, use the statement:

```
call sscal (5, 2.0, a(1,1), 6)
```

NOTE
The default vector argument is assumed to be 1.

Vector Arguments in Vector Math

Vector arguments of classic VM mathematical functions are passed in one-dimensional arrays with unit vector increment. It means that a vector of length *n* is passed contiguously in an array `a` whose values are defined as

`a(1), a(2), ..., a(n)`

.

Strided VM mathematical functions allow using positive increments for all input and output vector arguments.

To accommodate for arrays with other increments, or more complicated indexing, VM contains auxiliary Pack/Unpack functions that gather the array elements into a contiguous vector and then scatter them after the computation is complete.

Generally, if the vector elements are stored in a one-dimensional array *a* as

```
a(m1), a(m2), ..., a(mn)
```

and need to be regrouped into an array *y* as

```
y(k1), y(k2), ..., y(kn),
```

.

VM Pack/Unpack functions can use one of the following indexing methods:

Positive Increment Indexing

```
kj = 1 + incy * (j - 1), mj = 1 + inca * (j - 1), j = 1, ..., n.
```

.

Constraint: `incy > 0` and `inca > 0`.

For example, setting incy = 1 specifies gathering array elements into a contiguous vector.

This method is similar to that used in BLAS, with the exception that negative and zero increments are not permitted.

Index Vector Indexing

```
kj = iy(j), mj = ia(j), j = 1 ,..., n .
```

.

where ia and iy are arrays of length *n* that contain index vectors for the input and output arrays *a* and *y*, respectively.

Mask Vector Indexing

Indices `kj` , `mj` are such that:

```
my(kj) ≠ 0, ma(mj) ≠ 0 , j = 1,..., n .
```

.

where `ma` and `my` are arrays that contain mask vectors for the input and output arrays *a* and *y*, respectively.

Vector Mathematical Functions

Matrix Arguments

Matrix arguments of the Intel® oneAPI Math Kernel Library routines can be stored ineither one- or two-dimensional arrays, using the following storage schemes:

- conventional full storage(in a two-dimensional array)
- packed storage for Hermitian, symmetric, or triangular matrices (in a one-dimensional array)
- band storage for band matrices (in a two-dimensional array)
- rectangular full packed storage for symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels.

Full storage is the simplest scheme. A matrix *A* is stored in a two-dimensional array *a*, with the matrix element a_{ij} stored in the array element `a(i,j)` . , where *lda* is the leading dimension of array *a*.

If a matrix is triangular (upper or lower, as specified by the argument *uplo*), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set.

Routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangle of the matrix to be stored in the corresponding elements of the array:

if *uplo* ='U', a_{ij} is stored as described for $i \le j$, other elements of *a* need not be set.

if *uplo* ='L', a_{ij} is stored as described for $j \le i$, other elements of *a* need not be set.

Packed storage allows you to store symmetric, Hermitian, or triangular matrices more compactly: the relevant triangle (again, as specified by the argument *uplo*) is packed by columns in a one-dimensional array *ap*:

if *uplo* ='U', a_{ij} is stored in $ap(i + j(j - 1)/2)$ for $i \le j$

if *uplo* ='L', a_{ij} is stored in $ap(i + (2*n - j)*(j - 1)/2)$ for $j \le i$.

In descriptions of LAPACK routines, arrays with packed matrices have names ending in *p*.

Band storage is as follows: an *m*-by-*n* band matrix with *kl* non-zero sub-diagonals and *ku* non-zero super-diagonals is stored compactly in a two-dimensional array *ab* with *kl*+*ku* + 1 rows and *n* columns. Columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array . Thus,

a_{ij} is stored in *ab*(*ku*+1+*i*-*j*,*j*) for max(1,*j*-*ku*) ≤ *i* ≤ min(*n*,*j*+*kl*).

Use the band storage scheme only when *kl* and *ku* are much less than the matrix size *n*. Although the routines work correctly for all values of *kl* and *ku*, using the band storage is inefficient if your matrices are not really banded.

The band storage scheme is illustrated by the following example, when

$m = n = 6,\ kl = 2,\ ku = 1$

Array elements marked * are not used by the routines:

Banded matrix A

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

Band storage of A

$$\begin{matrix} * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{matrix}$$

When a general band matrix is supplied for *LU factorization*, space must be allowed to store *kl* additional super-diagonals generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with $kl + ku$ super-diagonals. Thus,

a_{ij} is stored in $ab(kl+ku+1+i-j,j)$ for $\max(1,j-ku) \le i \le \min(n,j+kl)$.

The band storage scheme for LU factorization is illustrated by the following example, when$m = n = 6,\ kl = 2,\ ku = 1$:

Banded matrix A

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

Band storage of A

*	*	*	+	+	+
*	*	+	+	+	+
*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*
a_{31}	a_{42}	a_{53}	a_{64}	*	*

Array elements marked * are not used by the routines; elements marked + need not be set on entry, but are required by the LU factorization routines to store the results. The input array will be overwritten on exit by the details of the LU factorization as follows:

*	*	*	u_{14}	u_{25}	u_{36}
*	*	u_{13}	u_{24}	u_{35}	u_{46}
*	u_{12}	u_{23}	u_{34}	u_{45}	u_{56}
u_{11}	u_{22}	u_{33}	u_{44}	u_{55}	u_{66}
m_{21}	m_{32}	m_{43}	m_{54}	m_{65}	*
m_{31}	m_{42}	m_{53}	m_{64}	*	*

where u_{ij} are the elements of the upper triangular matrix U, and m_{ij} are the multipliers used during factorization.

Triangular band matrices are stored in the same format, with either *kl*= 0 if upper triangular, or $ku = 0$ if lower triangular. For symmetric or Hermitian band matrices with *k* sub-diagonals or super-diagonals, you need to store only the upper or lower triangle, as specified by the argument *uplo*:

if *uplo* ='U', a_{ij} is stored in *ab*(*k*+1+*i*-*j*,*j*) for $\max(1,j-k) \le i \le j$

if *uplo* ='L', a_{ij} is stored in *ab*(1+*i*-*j*,*j*) for $j \le i \le \min(n,j+k)$.

In descriptions of LAPACK routines, arrays that hold matrices in band storage have names ending in *b*.

In Fortran, column-major ordering of storage is assumed. This means that elements of the same column occupy successive storage locations.

Three quantities are usually associated with a two-dimensional array argument: its leading dimension, which specifies the number of storage locations between elements in the same row, its number of rows, and its number of columns. For a matrix in full storage, the leading dimension of the array must be at least as large as the number of rows in the matrix.

A character transposition parameter is often passed to indicate whether the matrix argument is to be used in normal or transposed form or, for a complex matrix, if the conjugate transpose of the matrix is to be used.

The values of the transposition parameter for these three cases are the following:

'N' or *'n'*	normal (no conjugation, no transposition)
'T' or *'t'*	transpose
'C' or *'c'*	conjugate transpose.

Example. Two-Dimensional Complex Array

Suppose *A* (1:5, 1:4) is the complex two-dimensional array presented by matrix

$$\begin{bmatrix} (1.1,0.11) & (1.2,0.12) & (1.3,0.13) & (1.4,0.14) \\ (2.1,0.21) & (2.2,0.22) & (2.3,0.23) & (1.4,0.24) \\ (3.1,0.31) & (3.2,0.32) & (3.3,0.33) & (1.4,0.34) \\ (4.1,0.41) & (4.2,0.42) & (4.3,0.43) & (1.4,0.44) \\ (5.1,0.51) & (5.2,0.52) & (5.3,0.53) & (1.4,0.54) \end{bmatrix}$$

Let *transa* be the transposition parameter, *m* be the number of rows, *n* be the number of columns, and *lda* be the leading dimension. Then if

transa = *'N'*, *m* = 4, *n* = 2, and *lda* = 5, the matrix argument would be

$$\begin{bmatrix} (1.1,0.11) & (1.2,0.12) \\ (2.1,0.21) & (2.2,0.22) \\ (3.1,0.31) & (3.2,0.32) \\ (4.1,0.41) & (4.2,0.42) \end{bmatrix}$$

If *transa* = *'T'*, *m* = 4, *n* = 2, and *lda* =5, the matrix argument would be

$$\begin{bmatrix} (1.1,0.11) & (2.1,0.21) & (3.1,0.31) & (4.1,0.41) \\ (1.2,0.12) & (2.2,0.22) & (3.2,0.32) & (4.2,0.42) \end{bmatrix}$$

If *transa* = 'C', *m* = 4, *n* = 2, and *lda* =5, the matrix argument would be

$$\begin{bmatrix} (1.1,-0.11) & (2.1,-0.21) & (3.1,-0.31) & (4.1,-0.41) \\ (1.2,-0.12) & (2.2,-0.22) & (3.2,-0.32) & (4.2,-0.42) \end{bmatrix}$$

Note that care should be taken when using a leading dimension value which is different from the number of rows specified in the declaration of the two-dimensional array. For example, suppose the array *A* above is declared as `COMPLEX A (5,4)`.

Then if *transa* = 'N', *m* = 3, *n* = 4, and *lda* = 4, the matrix argument will be

$$\begin{bmatrix} (1.1,0.11) & (5.1,0.51) & (4.2,0.42) & (3.3,0.33) \\ (2.1,0.21) & (1.2,0.12) & (5.2,0.52) & (4.3,0.43) \\ (3.1,0.31) & (2.2,0.22) & (1.3,0.13) & (5.3,0.53) \end{bmatrix}$$

Rectangular Full Packed storage allows you to store symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels. To store an *n*-by-*n* triangle (and suppose for simplicity that *n* is even), you partition the triangle into three parts: two *n*/2-by-*n*/2 triangles and an *n*/2-by-*n*/2 square, then pack this as an *n*-by-*n*/2 rectangle (or *n*/2-by-*n* rectangle), by transposing (or transpose-conjugating) one of the triangles and packing it next to the other triangle. Since the two triangles are stored in full storage, you can use existing efficient routines on them.

There are eight cases of RFP storage representation: when *n* is even or odd, the packed matrix is transposed or not, the triangular matrix is lower or upper. See below for all the eight storage schemes illustrated:

n is odd, *A* is lower triangular

Full format

$$\begin{matrix} a_{11} & X & X & X & X & X & X \\ a_{21} & a_{22} & X & X & X & X & X \\ a_{31} & a_{32} & a_{33} & X & X & X & X \\ a_{41} & a_{42} & a_{43} & a_{44} & X & X & X \\ \mathbf{a_{51}} & \mathbf{a_{52}} & \mathbf{a_{53}} & \mathbf{a_{54}} & a_{55} & X & X \\ \mathbf{a_{61}} & \mathbf{a_{62}} & \mathbf{a_{63}} & \mathbf{a_{64}} & a_{65} & a_{66} & X \\ \mathbf{a_{71}} & \mathbf{a_{72}} & \mathbf{a_{73}} & \mathbf{a_{74}} & a_{75} & a_{76} & a_{77} \end{matrix}$$

RFP (not transposed)

$$\begin{matrix} a_{11} & a_{55} & a_{65} & a_{75} \\ a_{21} & a_{22} & a_{66} & a_{76} \\ a_{31} & a_{32} & a_{33} & a_{77} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ \mathbf{a_{51}} & \mathbf{a_{52}} & \mathbf{a_{53}} & \mathbf{a_{54}} \\ \mathbf{a_{61}} & \mathbf{a_{62}} & \mathbf{a_{63}} & \mathbf{a_{64}} \\ \mathbf{a_{71}} & \mathbf{a_{72}} & \mathbf{a_{73}} & \mathbf{a_{74}} \end{matrix}$$

RFP (transposed)

$$\begin{matrix} a_{11} & a_{21} & a_{31} & a_{41} & \mathbf{a_{51}} & \mathbf{a_{61}} & \mathbf{a_{71}} \\ a_{55} & a_{22} & a_{32} & a_{42} & \mathbf{a_{52}} & \mathbf{a_{62}} & \mathbf{a_{72}} \\ a_{65} & a_{66} & a_{33} & a_{43} & \mathbf{a_{53}} & \mathbf{a_{63}} & \mathbf{a_{73}} \\ a_{75} & a_{76} & a_{77} & a_{44} & \mathbf{a_{54}} & \mathbf{a_{64}} & \mathbf{a_{74}} \end{matrix}$$

n is even, *A* is lower triangular

Full format

$$\begin{matrix} a_{11} & X & X & X & X & X \\ a_{21} & a_{22} & X & X & X & X \\ a_{31} & a_{32} & a_{33} & X & X & X \\ \mathbf{a_{41}} & \mathbf{a_{42}} & \mathbf{a_{43}} & a_{44} & X & X \\ \mathbf{a_{51}} & \mathbf{a_{52}} & \mathbf{a_{53}} & a_{54} & a_{55} & X \\ \mathbf{a_{61}} & \mathbf{a_{62}} & \mathbf{a_{63}} & a_{64} & a_{65} & a_{66} \end{matrix}$$

RFP (not transposed)

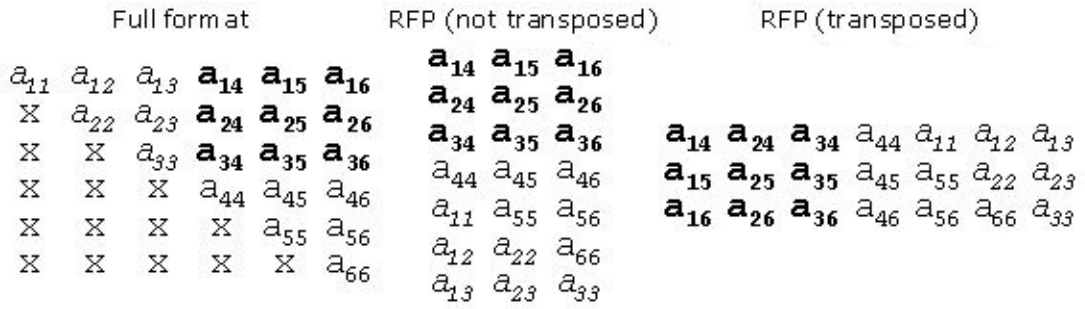
$$\begin{matrix} a_{44} & a_{54} & a_{64} \\ a_{11} & a_{55} & a_{65} \\ a_{21} & a_{22} & a_{66} \\ a_{31} & a_{32} & a_{33} \\ \mathbf{a_{41}} & \mathbf{a_{42}} & \mathbf{a_{43}} \\ \mathbf{a_{51}} & \mathbf{a_{52}} & \mathbf{a_{53}} \\ \mathbf{a_{61}} & \mathbf{a_{62}} & \mathbf{a_{63}} \end{matrix}$$

RFP (transposed)

$$\begin{matrix} a_{44} & a_{11} & a_{21} & a_{31} & \mathbf{a_{41}} & \mathbf{a_{51}} & \mathbf{a_{61}} \\ a_{54} & a_{55} & a_{22} & a_{32} & \mathbf{a_{42}} & \mathbf{a_{52}} & \mathbf{a_{62}} \\ a_{64} & a_{65} & a_{66} & a_{33} & \mathbf{a_{43}} & \mathbf{a_{53}} & \mathbf{a_{63}} \end{matrix}$$

n is odd, *A* is upper triangular

n is even, *A* is upper triangular



Intel® oneAPI Math Kernel Library (oneMKL) provides a number of routines such as`?hfrk`, `?sfrk` performing BLAS operations working directly on RFP matrices, as well as some conversion routines, for instance, `?tpttf` goes from the standard packed format to RFP and `?trttf` goes from the full format to RFP.

Please refer to the Netlib site for more information.

Note that in the descriptions of LAPACK routines, arrays with RFP matrices have names ending in `fp`.

Appendix C: Specific Features of Fortran 95 Interfaces for LAPACK Routines

Intel® MKL implements Fortran 95 interface for LAPACK package, further referred to as MKL LAPACK95, to provide full capacity of MKL FORTRAN 77 LAPACK routines. This is the principal difference of Intel MKL from the Netlib Fortran 95 implementation for LAPACK.

A new feature of MKL LAPACK95 by comparison with Intel MKL LAPACK77 implementation is presenting a package of source interfaces along with wrappers that make the implementation compiler-independent. As a result, the MKL LAPACK package can be used in all programming environments intended for Fortran 95.

Depending on the degree and type of difference from Netlib implementation, the MKL LAPACK95 interfaces fall into several groups that require different transformations (see "MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib Implementation"). The groups are given in full with the calling sequences of the routines and appropriate differences from Netlib analogs.

The following conventions are used:

```
<interface> ::=  <name of interface> '(' <arguments list>')'
<arguments list> ::=  <first argument> {<argument>}*
<first argument> ::=  < identifier >
<argument> ::=  <required argument>|<optional argument>
<required argument> ::=  ',' <identifier>
<optional argument> ::=  '[,' <identifier> ']'
<name of interface> ::=  <identifier>
```

where defined notions are separated from definitions by `::=`, notion names are marked by angle brackets, terminals are given in quotes, and `{…}*` denotes repetition zero, one, or more times.

<first argument> and each *<required argument>* should be present in all calls of denoted interface, *<optional argument>* may be omitted. Comments to interface definitions are provided where necessary. Comment lines begin with character `!`.

Two interfaces with one name are presented when two variants of subroutine calls (separated by types of arguments) exist.

Appendix D: FFTW Interface to Intel® Math Kernel Library

Intel® oneAPI Math Kernel Library (oneMKL) offers FFTW2 and FFTW3 interfaces to Intel® oneAPI Math Kernel Library (oneMKL) Fast Fourier Transform and Trigonometric Transform functionality. The purpose of these interfaces is to enable applications using FFTW (www.fftw.org) to gain performance with Intel® oneAPI Math Kernel Library (oneMKL) without changing the program source code.

Both FFTW2 and FFTW3 interfaces are provided in open source as FFTW wrappers to Intel® oneAPI Math Kernel Library (oneMKL). For ease of use, FFTW3 interface is also integrated in Intel® oneAPI Math Kernel Library (oneMKL).

FFTW Notational Conventions

This appendix typically employs path notations for Windows* OS.

FFTW2 Interface to Intel® oneAPI Math Kernel Library

This section describes a collection of C and Fortran wrappers providing FFTW 2.x interface to Intel® oneAPI Math Kernel Library (oneMKL). The wrappers translate calls to FFTW 2.x functions into the calls of the Intel® oneAPI Math Kernel Library (oneMKL) Fast Fourier Transform interface (FFT interface).

Note that Intel® oneAPI Math Kernel Library (oneMKL) FFT interface operates on both single- and double-precision floating-point data types.

Because of differences between FFTW and Intel® oneAPI Math Kernel Library (oneMKL) FFT functionalities, there are restrictions on using wrappers instead of the FFTW functions. Some FFTW functions have empty wrappers. However, many typical FFTs can be computed using these wrappers.

Refer to Fourier Transform Functions, for better understanding the effects from the use of the wrappers.

Wrappers Reference

The section provides a brief reference for the FFTW 2.x C interface. For details please refer to the original FFTW 2.x documentation available at www.fftw.org.

Each FFTW function has its own wrapper. Some of them, which are *not* expressly listed in this section, are empty and do nothing, but they are provided to avoid link errors and satisfy the function calls.

See Also

Limitations of the FFTW2 Interface to Intel® oneAPI Math Kernel Library (oneMKL)

One-dimensional Complex-to-complex FFTs

The following functions compute a one-dimensional complex-to-complex Fast Fourier transform.

```
fftw_plan fftw_create_plan(int n, fftw_direction dir, int flags);
```

```
fftw_plan fftw_create_plan_specific(int n, fftw_direction dir, int flags, fftw_complex
*in, int istride, fftw_complex *out, int ostride);
```

```
void fftw(fftw_plan plan, int howmany, fftw_complex *in, int istride, int idist,
fftw_complex *out, int ostride, int odist);
```

```
void fftw_one(fftw_plan plan, fftw_complex *in , fftw_complex *out);

void fftw_destroy_plan(fftw_plan plan);
```

Multi-dimensional Complex-to-complex FFTs

The following functions compute a multi-dimensional complex-to-complex Fast Fourier transform.

```
fftwnd_plan fftwnd_create_plan(int rank, const int *n, fftw_direction dir, int flags);

fftwnd_plan fftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);

fftwnd_plan fftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int flags);

fftwnd_plan fftwnd_create_plan_specific(int rank, const int *n, fftw_direction dir, int
flags, fftw_complex *in, int istride, fftw_complex *out, int ostride);

fftwnd_plan fftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int flags,
fftw_complex *in, int istride, fftw_complex *out, int ostride);

fftwnd_plan fftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir, int
flags, fftw_complex *in, int istride, fftw_complex *out, int ostride);

void fftwnd(fftwnd_plan plan, int howmany, fftw_complex *in, int istride, int idist,
fftw_complex *out, int ostride, int odist);

void fftwnd_one(fftwnd_plan plan, fftw_complex *in, fftw_complex *out);

void fftwnd_destroy_plan(fftwnd_plan plan);
```

One-dimensional Real-to-half-complex/Half-complex-to-real FFTs

Half-complex representation of a conjugate-even symmetric vector of size *N* in a real array of the same size *N* consists of *N*/2+1 real parts of the elements of the vector followed by non-zero imaginary parts in the reverse order. Because the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface does not currently support this representation, all wrappers of this kind are empty and do nothing.

Nevertheless, you can perform one-dimensional real-to-complex and complex-to-real transforms using `rfftwnd` functions with *rank*=1.

See Also

Multi-dimensional Real-to-complex/complex-to-real FFTs

Multi-dimensional Real-to-complex/Complex-to-real FFTs

The following functions compute multi-dimensional real-to-complex and complex-to-real Fast Fourier transforms.

```
rfftwnd_plan rfftwnd_create_plan(int rank, const int *n, fftw_direction dir, int
flags);

rfftwnd_plan rfftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);

rfftwnd_plan rfftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int
flags);

rfftwnd_plan rfftwnd_create_plan_specific(int rank, const int *n, fftw_direction dir,
int flags, fftw_real *in, int istride, fftw_real *out, int ostride);

rfftwnd_plan rfftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int
flags, fftw_real *in, int istride, fftw_real *out, int ostride);

rfftwnd_plan rfftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir,
int flags, fftw_real *in, int istride, fftw_real *out, int ostride);
```

```
void rfftwnd_real_to_complex(rfftwnd_plan plan, int howmany, fftw_real *in, int
istride, int idist, fftw_complex *out, int ostride, int odist);
void rfftwnd_complex_to_real(rfftwnd_plan plan, int howmany, fftw_complex *in, int
istride, int idist, fftw_real *out, int ostride, int odist);
void rfftwnd_one_real_to_complex(rfftwnd_plan plan, fftw_real *in, fftw_complex *out);
void rfftwnd_one_complex_to_real(rfftwnd_plan plan, fftw_complex *in, fftw_real *out);
void rfftwnd_destroy_plan(rfftwnd_plan plan);
```

Multi-threaded FFTW

Unlike the original FFTW interface, every computational function in the FFTW2 interface to Intel® oneAPI Math Kernel Library (oneMKL) provides multithreaded computation by default, with the maximum number of threads permitted in FFT functions (see "Techniques to Set the Number of Threads" in *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*). To limit the number of threads, call the threaded FFTW computational functions:

```
void fftw_threads(int nthreads, fftw_plan plan, int howmany, fftw_complex *in, int
istride, int idist, fftw_complex *out, int ostride, int odist);
void fftw_threads_one(int nthreads, rfftwnd_plan plan, fftw_complex *in, fftw_complex
*out);
...
void rfftwnd_threads_real_to_complex( int nthreads, rfftwnd_plan plan, int howmany,
fftw_real *in, int istride, int idist, fftw_complex *out, int ostride, int odist);
```

Compared to its non-threaded counterpart, every threaded computational function has `threads_` as the second part of its name and additional first parameter *nthreads*. Set the *nthreads* parameter to the thread limit to ensure that the computation requires at most that number of threads.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

FFTW Support Functions

The FFTW wrappers provide memory allocation functions to be used with FFTW:

```
void* fftw_malloc(size_t n);
void fftw_free(void* x);
```

The `fftw_malloc` wrapper aligns the memory on a 16-byte boundary.

If `fftw_malloc` fails to allocate memory, it aborts the application. To override this behavior, set a global variable `fftw_malloc_hook` and optionally the complementary variable `fftw_free_hook`:

```
void *(*fftw_malloc_hook) (size_t n);
void (*fftw_free_hook) (void *p);
```

The wrappers use the function `fftw_die` to abort the application in cases when a caller cannot be informed of an error otherwise (for example, in computational functions that return `void`). To override this behavior, set a global variable `fftw_die_hook`:

```
void (*fftw_die_hook)(const char *error_string);
void fftw_die(const char *s);
```

Calling FFTW2 Interface Wrappers from Fortran

The FFTW2 wrappers to Intel® oneAPI Math Kernel Library (oneMKL) provide the following subroutines for calling from Fortran:

```
call fftw_f77_create_plan(plan, n, dir, flags)
call fftw_f77(plan, howmany, in, istride, idist, out, ostride, odist)
call fftw_f77_one(plan, in, out)
call fftw_f77_threads(nthreads, plan, howmany, in, istride, idist, out, ostride, odist)
call fftw_f77_threads_one(nthreads, plan, in, out)
call fftw_f77_destroy_plan(plan)

call fftwnd_f77_create_plan(plan, rank, n, dir, flags)
call fftw2d_f77_create_plan(plan, nx, ny, dir, flags)
call fftw3d_f77_create_plan(plan, nx, ny, nz, dir, flags)
call fftwnd_f77(plan, howmany, in, istride, idist, out, ostride, odist)
call fftwnd_f77_one(plan, in, out)
call fftwnd_f77_threads(nthreads, plan, howmany, in, istride, idist, out, ostride,
odist)
call fftwnd_f77_threads_one(nthreads, plan, in, out)
call fftwnd_f77_destroy_plan(plan)

call rfftw_f77_create_plan(plan, n, dir, flags)
call rfftw_f77(plan, howmany, in, istride, idist, out, ostride, odist)
call rfftw_f77_one(plan, in, out)
call rfftw_f77_threads(nthreads, plan, howmany, in, istride, idist, out, ostride, odist)
call rfftw_f77_threads_one(nthreads, plan, in, out)
call rfftw_f77_destroy_plan(plan)

call rfftwnd_f77_create_plan(plan, rank, n, dir, flags)
call rfftw2d_f77_create_plan(plan, nx, ny, dir, flags)
call rfftw3d_f77_create_plan(plan, nx, ny, nz, dir, flags)
call rfftwnd_f77_complex_to_real(plan, howmany, in, istride, idist, out, ostride, odist)
call rfftwnd_f77_one_complex_to_real (plan, in, out)
call rfftwnd_f77_real_to_complex(plan, howmany, in, istride, idist, out, ostride, odist)
call rfftwnd_f77_one_real_to_complex (plan, in, out)
call rfftwnd_f77_threads_complex_to_real(nthreads, plan, howmany, in, istride, idist,
out, ostride, odist)
call rfftwnd_f77_threads_one_complex_to_real(nthreads, plan, in, out)
call rfftwnd_f77_threads_real_to_complex(nthreads, plan, howmany, in, istride, idist,
out, ostride, odist)
call rfftwnd_f77_threads_one_real_to_complex(nthreads, plan, in, out)
call rfftwnd_f77_destroy_plan(plan)

call fftw_f77_threads_init(info)
```

The FFTW Fortran functions are wrappers to FFTW C functions.

See also these resources:

www.fftw.org	for the original FFTW 2.x documentation.
Limitations of the FFTW2 Interface to Intel MKL	for limitations of the wrappers.

Limitations of the FFTW2 Interface to Intel® oneAPI Math Kernel Library (oneMKL)

The FFTW2 wrappers implement the functionality of only those FFTW functions that Intel® oneAPI Math Kernel Library (oneMKL) can reasonably support. Other functions are provided as no-operation functions, whose only purpose is to satisfy link-time symbol resolution. Specifically, no-operation functions include:

- Real-to-half-complex and respective backward transforms
- Print plan functions
- Functions for importing/exporting/forgetting wisdom
- Most of the FFTW functions not covered by the original FFTW2 documentation

Because the Intel® oneAPI Math Kernel Library (oneMKL) implementation of FFTW2 wrappers does not use plan and plan node structures declared in `fftw.h`, the behavior of an application that relies on the internals of the plan structures defined in that header file is undefined.

FFTW2 wrappers define plan as a set of attributes, such as strides, used to commit the Intel® oneAPI Math Kernel Library (oneMKL) FFT descriptor structure. If an FFTW2 computational function is called with attributes different from those recorded in the plan, the function attempts to adjust the attributes of the plan and recommit the descriptor. So, repeated calls of a computational function with the same plan but different strides, distances, and other parameters may be performance inefficient.

Plan creation functions disregard most planner flags passed through the *flags* parameter. These functions take into account only the following values of *flags*:

- `FFTW_IN_PLACE`

  If this value of *flags* is supplied, the plan is marked so that computational functions using that plan ignore the parameters related to output (*out*, *ostride*, and *odist*). Unlike the original FFTW interface, the wrappers never use the *out* parameter as a scratch space for in-place transforms.
- `FFTW_THREADSAFE`

  If this value of *flags* is supplied, the plan is marked read-only. An attempt to change attributes of a read-only plan aborts the application.

FFTW wrappers are generally not thread safe. Therefore, do not use the same plan in parallel user threads simultaneously.

Installing FFTW2 Interface Wrappers

Wrappers are delivered as source code, which you must compile to build the wrapper library. Then you can substitute the wrapper and Intel® oneAPI Math Kernel Library (oneMKL) libraries for the FFTW library. The source code for the wrappers, makefiles, and files with lists of wrappers are located in the `.\interfaces\fftw2xf`subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

Creating the Wrapper Library

Three header files are used to compile the Fortran wrapper library: `fftw2_mkl.h`, `fftw2_f77_mkl.h`, and `fftw.h`. The `fftw2_mkl.h` and `fftw2_f77_mkl.h` files are located in the `.\interfaces\fftw2xf\wrappers`subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

The file `fftw.h`, used to compile libraries and located in the `.\include\fftw`subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory, slightly differs from the original FFTW (www.fftw.org) header file`fftw.h`.

The source code for the wrappers, makefiles, and files with lists of functions are located in the `.\interfaces\fftw2xf` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

A wrapper library contains wrappers for complex and real transforms in a serial and multi-threaded mode for double- or single-precision floating-point data types. A makefile parameter manages the data type.

Parameters of a makefile also specify the platform (required), compiler, and data precision. The makefile comment heading provides the exact description of these parameters.

Because a C compiler builds the Fortran wrapper library, function names in the wrapper library and Fortran object module may be different. The file `fftw2_f77_mkl.h` in the `.\interfaces\fftw2xf \source`subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory defines function names according to the names in the Fortran module. If a required name is missing in the file, you can modify the file to add the name before building the library.

To build the library, run the `make` command on Linux* OS and macOS* or the `nmake` command on Windows* OS with appropriate parameters.

For example, on Linux OS the command

```
make libintel64
```

builds a double-precision wrapper library for Intel® 64 architecture based applications using the Intel® oneAPI DPC++/C++ Compiler or the Intel® Fortran Compiler (Compilers and data precision are chosen by default.)

Each makefile creates the library in the directory with Intel® oneAPI Math Kernel Library (oneMKL) libraries corresponding to the platform used. For example, `./lib/ia32` (on Linux OS and macOS) or `.\lib\ia32` (on Windows* OS).

In the names of a wrapper library, the suffix corresponds to the compiler used and the letter preceding the underscore is `"f"` for the Fortran programming language.

For example,

`fftw2xf_intel.lib` (on Windows OS); `libfftw2xf_intel.a` (on Linux OS and macOS);

Application Assembling

Use the necessary original FFTW (www.fftw.org) header files without any modifications. Use the created wrapper library and the Intel® oneAPI Math Kernel Library (oneMKL) library instead of the FFTW library.

Running FFTW2 Interface Wrapper Examples

Intel® oneAPI Math Kernel Library (oneMKL) provides examples to demonstrate how to use the MPI FFTW wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw2xf` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory . To build examples, several additional files are needed: `fftw.h`, `fftw_threads.h`, `rfftw.h`, `rfftw_threads.h`, and `fftw_f77.i`. These files are distributed with permission from FFTW and are available in `.\include\fftw`. The original files can also be found in FFTW 2.1.5 at http://www.fftw.org/download.html.

An example makefile uses the `function` parameter in addition to the parameters of the corresponding wrapper library makefile (see Creating a Wrapper Library). The makefile comment heading provides the exact description of these parameters.

An example makefile normally invokes examples. However, if the appropriate wrapper library is not yet created, the makefile first builds the library the same way as the wrapper library makefile does and then proceeds to examples.

If the parameter `function=<example_name>` is defined, only the specified example runs. Otherwise, all examples from the appropriate subdirectory run. The subdirectory `.\_results` is created, and the results are stored there in the `<example_name>.res` files.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Product and Performance Information
Notice revision #20201201

FFTW3 Interface to Intel® oneAPI Math Kernel Library

This section describes a collection of FFTW3 wrappers to Intel® oneAPI Math Kernel Library (oneMKL). The wrappers translate calls of FFTW3 functions to the calls of the Intel® oneAPI Math Kernel Library (oneMKL) Fourier transform (FFT) or Trigonometric Transform (TT) functions. The purpose of FFTW3 wrappers is to enable developers whose programs currently use the FFTW3 library to gain performance with the Intel® oneAPI Math Kernel Library (oneMKL) Fourier transforms without changing the program source code.

The FFTW3 wrappers provide a limited functionality compared to the original FFTW 3.x library, because of differences between FFTW and Intel® oneAPI Math Kernel Library (oneMKL) FFT and TT functionality. This section describes limitations of the FFTW3 wrappers and hints for their usage. Nevertheless, many typical FFT tasks can be performed using the FFTW3 wrappers to Intel® oneAPI Math Kernel Library (oneMKL).

The FFTW3 wrappers are integrated in Intel® oneAPI Math Kernel Library (oneMKL). The only change required to use Intel® oneAPI Math Kernel Library (oneMKL) through the FFTW3 wrappers is to link your application using FFTW3 against Intel® oneAPI Math Kernel Library (oneMKL).

A reference implementation of the FFTW3 wrappers is also provided in open source. You can find it in the `interfaces` directory of the Intel® oneAPI Math Kernel Library (oneMKL) distribution. You can use the reference implementation to create your own wrapper library (see Building Your Own Wrapper Library)

See also these resources:

Intel® oneAPI Math Kernel Library (oneMKL) Release Notes	for the version of the FFTW3 library supported by the wrappers.
www.fftw.org	for a description of the FFTW interface.
Fourier Transform Functions	for a description of the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface.
Trigonometric Transform Routines	for a description of Intel® oneAPI Math Kernel Library (oneMKL) TT interface.

Using FFTW3 Wrappers

The FFTW3 wrappers are a set of functions and data structures depending on one another. The wrappers are not designed to provide the interface on a function-per-function basis. Some FFTW3 wrapper functions are empty and do nothing, but they are present to avoid link errors and satisfy function calls.

This document does not list the declarations of the functions that the FFTW3 wrappers provide (you can find the declarations in the `fftw3.h` header file). Instead, this section comments on particular limitations of the wrappers and provides usage hints:. These are some known limitations of FFTW3 wrappers and their usage in Intel® oneAPI Math Kernel Library (oneMKL).

- The FFTW3 wrappers do not support long double precision because Intel® oneAPI Math Kernel Library (oneMKL) FFT functions operate only on single- and double-precision floating-point data types. Therefore the functions with prefix `fftwl_`, supporting the `long double` data type, are not provided.
- The wrappers provide equivalent implementation for double- and single-precision functions (those with prefixes `fftw_` and `fftwf_`, respectively). So, all these comments equally apply to the double- and single-precision functions and will refer to functions with prefix `fftw_`, that is, double-precision functions, for brevity.
- The FFTW3 interface that the wrappers provide is defined in the `fftw3.h` and `fftw3.f` header files. These files are borrowed from the FFTW3.x package and distributed within Intel® oneAPI Math Kernel Library (oneMKL) with permission. Additionally, the `fftw3_mkl.h`, `fftw3_mkl.f`, and `fftw3_mkl_f77.h` header files define supporting structures and supplementary constants and macros as well as expose Fortran interface in C.

- Actual functionality of the plan creation wrappers is implemented in guru64 set of functions. Basic interface, advanced interface, and guru interface plan creation functions call the guru64 interface functions. So, all types of the FFTW3 plan creation interface in the wrappers are functional.
- Plan creation functions may return a `NULL` plan, indicating that the functionality is not supported. So, please carefully check the result returned by plan creation functions in your application. In particular, the following problems return a `NULL` plan:
  - c2r and r2c problems with a split storage of complex data.
  - r2r problems with *kind* values `FFTW_R2HC`, `FFTW_HC2R`, and `FFTW_DHT`. The only supported r2r kinds are even/odd DFTs (sine/cosine transforms).
  - Multidimensional r2r transforms.
  - Transforms of multidimensional vectors. That is, the only supported values for parameter *howmany_rank* in guru and guru64 plan creation functions are 0 and 1.
  - Multidimensional transforms with *rank* > MKL_MAXRANK.
- The `MKL_RODFT00` value of the *kind* parameter is introduced by the FFTW3 wrappers. For better performance, you are strongly encouraged to use this value rather than `FFTW_RODFT00`. To use this *kind* value, provide an extra first element equal to 0.0 for the input/output vectors. Consider the following example:

```
plan1 = fftw_plan_r2r_1d(n, in1, out1, FFTW_RODFT00, FFTW_ESTIMATE);
plan2 = fftw_plan_r2r_1d(n, in2, out2, MKL_RODFT00, FFTW_ESTIMATE);
```

  Both plans perform the same transform, except that the *in2/out2* arrays have one extra zero element at location 0. For example, if *n*=3, *in1*={*x,y,z*} and *out1*={*u,v,w*}, then *in2*={0,*x,y,z*} and *out2*={0,*u,v,w*}.
- The *flags* parameter in plan creation functions is always ignored. The same algorithm is used regardless of the value of this parameter. In particular, *flags* values `FFTW_ESTIMATE`, `FFTW_MEASURE`, etc. have no effect.
- For multithreaded plans, use normal sequence of calls to the `fftw_init_threads()` and `fftw_plan_with_nthreads()` functions (refer to FFTW documentation).
- Memory allocation function `fftw_malloc` returns memory aligned at a 16-byte boundary. You must free the memory with `fftw_free`.
- Fortran wrappers (see Calling Wrappers from Fortran) use the `INTEGER` type, which is 32-bit in LP64 interfaces and 64-bit in ILP64 interfaces.
- The wrappers typically indicate a problem by returning a `NULL` plan. In a few cases, the wrappers may report a descriptive message of the problem detected. By default the reporting is turned off. To turn it on, set variable `fftw3_mkl.verbose` to a non-zero value, for example:

```
#include "fftw3.h"
#include "fftw3_mkl.h"
fftw3_mkl.verbose = 0;
plan = fftw_plan_r2r(...);
```

- The following functions are empty:
  - For saving, loading, and printing plans
  - For saving and loading wisdom
  - For estimating arithmetic cost of the transforms.
- Do not use macro `FFTW_DLL` with the FFTW3 wrappers to Intel® oneAPI Math Kernel Library (oneMKL).
- Do not use negative stride values. Though FFTW3 wrappers support negative strides in the part of advanced and guru FFTW interface, the underlying implementation does not.
- Do not set a FFTW2 wrapper library before a FFTW3 wrapper library or Intel® oneAPI Math Kernel Library (oneMKL) in your link line application. All libraries define "fftw_destroy_plan" symbol and linkage in incorrect order results into expected errors.

Calling FFTW3 Interface Wrappers from Fortran

Intel® oneAPI Math Kernel Library (oneMKL) also provides Fortran 77 interfaces of the FFTW3 wrappers. The Fortran wrappers are available for all FFTW3 interface functions and are based on C interface of the FFTW3 wrappers. Therefore they have the same functionality and restrictions as the corresponding C interface wrappers.

The Fortran wrappers use the default `INTEGER` type for integer arguments. The default `INTEGER`is 32-bit in Intel® oneAPI Math Kernel Library (oneMKL) LP64 interfaces and 64-bit in ILP64 interfaces. Argument*`plan`* in a Fortran application must have type `INTEGER*8`.

The wrappers that are double-precision subroutines have prefix `dfftw_`, single-precision subroutines have prefix `sfftw_` and provide an equivalent functionality. Long double subroutines (with prefix `lfftw_`) are not provided.

The Fortran FFTW3 wrappers use the default Intel® Fortran compiler convention for name decoration. If your compiler uses a different convention, or if you are using compiler options affecting the name decoration (such as `/Qlowercase`), you may need to compile the wrappers from sources, as described in section Building Your Own Wrapper Library.

For interoperability with C, the declaration of the Fortran FFTW3 interface is provided in header file `include/fftw/fftw3_mkl_f77.h`.

You can call Fortran wrappers from a FORTRAN 77 or Fortran 90 application, although Intel® oneAPI Math Kernel Library (oneMKL) does not provide a Fortran 90 module for the wrappers. For a detailed description of the FFTW Fortran interface, refer to FFTW3 documentation (www.fftw.org).

The following example illustrates calling the FFTW3 wrappers from Fortran:

```
INTEGER*8 plan
INTEGER N
INCLUDE 'fftw3.f'
COMPLEX*16 IN(*), OUT(*)
!...initialize array IN
CALL DFFTW_PLAN_DFT_1D(PLAN, N, IN, OUT, -1, FFTW_ESTIMATE)
IF (PLAN .EQ. 0) STOP
CALL DFFTW_EXECUTE
!...result is in array OUT
```

Building Your Own FFTW3 Interface Wrapper Library

The FFTW3 wrappers to Intel® oneAPI Math Kernel Library (oneMKL) are delivered both integrated in Intel® oneAPI Math Kernel Library (oneMKL) and as source code, which can be compiled to build a standalone wrapper library with exactly the same functionality. Normally you do not need to build the wrappers yourself.However, if your Fortran application is compiled with a compiler that uses a different name decoration than the Intel® Fortran compiler or if you are using compiler options altering the Fortran name decoration, you may need to build the wrappers that use the appropriate name changing convention.

The source code for the wrappers, makefiles, and files with lists of functions are located in the `.\interfaces\fftw3xf` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS and macOS* or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target `libia32` or `libintel64` defines the platform architecture, and the other parameters specify the compiler, size of the default integer type, and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the FFTW3 Fortran wrappers to Intel® oneAPI Math Kernel Library (oneMKL) for use from the GNU g77* Fortran compiler on Linux OS based on Intel® 64 architecture:

```
cd interfaces/fftw3xf
make libintel64 compiler=gnu fname=a_name__ INSTALL_DIR=/my/path
```

This command builds the wrapper library using the GNU gcc compiler, decorates the name with the second underscore, and places the result, named `libfftw3xf_gnu.a`, into the `/my/path` directory. The name of the resulting library is composed of the name of the compiler used and may be changed by an optional parameter `INSTALL_LIBNAME`.

Building an Application With FFTW3 Interface Wrappers

Normally, the only change needed to build your application with FFTW3 wrappers replacing original FFTW library is to add Intel® oneAPI Math Kernel Library (oneMKL) at the link stage (see section *"Linking Your Application with Intel® oneAPI Math Kernel Library" in the Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*).

If you recompile your application, add subdirectory `include\fftw` to the search path for header files to avoid FFTW3 version conflicts.

Sometimes, you may have to modify your application according to the following recommendations:

- The application requires

  `#include "fftw3.h"`,

  which it probably already includes.
- The application does not require

  `#include "mkl_dfti.h"` .
- The application does not require

  `#include "fftw3_mkl.h"` .

  It is required only in case you want to use the `MKL_RODFT00` constant.
- If the application does not check whether a `NULL` plan is returned by plan creation functions, this check must be added, because the FFTW3 to Intel® oneAPI Math Kernel Library (oneMKL) wrappers do not provide 100% of FFTW3 functionality.

Running FFTW3 Interface Wrapper Examples

There are some examples that demonstrate how to use the wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw3xf` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory. To build Fortran examples, one additional file `fftw3.f` is needed. This file is distributed with permission from FFTW and is available in the `.\include\fftw`subdirectory of the Intel® oneAPI Math Kernel Library (oneMKL) directory. The original file can also be found in FFTW 3.3.4 at http://www.fftw.org/download.html.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. If the parameter `function=<example_name>` is defined, then only the specified example will run. Otherwise, all examples will be executed. Results of running the examples are saved in subdirectory `.\_results` in files with extension `.res`.

For detailed information about options for the example makefile, refer to the makefile.

MPI FFTW3 Wrappers

This section describes a collection of MPI FFTW wrappers to Intel® oneAPI Math Kernel Library (oneMKL).

MPI FFTW wrappers are available only with Intel® oneAPI Math Kernel Library (oneMKL) for the Linux* and Windows* operating systems.

These wrappers translate calls of MPI FFTW functions to the calls of the Intel® oneAPI Math Kernel Library (oneMKL) cluster Fourier transform (CFFT) functions. The purpose of the wrappers is to enable users of MPI FFTW functions improve performance of the applications without changing the program source code.

Although the MPI FFTW wrappers provide less functionality than the original FFTW3 because of differences between MPI FFTW and Intel® oneAPI Math Kernel Library (oneMKL) CFFT, the wrappers cover many typical CFFT use cases.

The MPI FFTW wrappers are provided as source code. To use the wrappers, you need to build your own wrapper library (see Building Your Own Wrapper Library).

See also these resources:

Intel® oneAPI Math Kernel Library (oneMKL) Release Notes	for the version of the FFTW3 library supported by the wrappers.
www.fftw.org	for a description of the MPI FFTW interface.
Cluster FFT Functions	for a description of the Intel® oneAPI Math Kernel Library (oneMKL) CFFT interface.

Building Your Own Wrapper Library

The MPI FFTW wrappers for FFTW3 are delivered as source code, which can be compiled to build a wrapper library.

The source code for the wrappers, makefiles, and files with lists of functions are located in subdirectory `.\interfaces\fftw3x_cdft` in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target `libia32` or `libintel64` defines the platform architecture, and the other parameters specify the compiler, size of the default `INTEGER` type, as well as the name and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the MPI FFTW wrappers to Intel® oneAPI Math Kernel Library (oneMKL) for use from the GNU C compiler on Linux OS based on Intel® 64 architecture:

```
cd interfaces/fftw3x_cdft
make libintel64 compiler=gnu mpi=openmpi INSTALL_DIR=/my/path
```

This command builds the wrapper library using the GNU gcc compiler so that the final executable can use Open MPI, and places the result, named `libfftw3x_cdft_DOUBLE.a`, into directory `/my/path`.

Building an Application

Normally, the only change needed to build your application with MPI FFTW wrappers replacing original FFTW3 library is to add Intel® oneAPI Math Kernel Library (oneMKL) and the wrapper library at the link stage (see section "*Linking Your Application with Intel® oneAPI Math Kernel Library*" in the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*).

When you are recompiling your application, add subdirectory `include\fftw` to the search path for header files to avoid FFTW3 version conflicts.

Running Examples

There are some examples that demonstrate how to use the MPI FFTW wrapper library for FFTW3. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw3xf_cdft` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. Results of running the examples are saved in subdirectory `.\_results` in files with extension `.res`.

For detailed information about options for the example makefile, refer to the makefile.

See Also

Building Your Own Wrapper Library

Appendix E: Code Examples

This appendix presents code examples of using some Intel® oneAPI Math Kernel Library (oneMKL) routines and functions.

Please refer to respective sections in the document for detailed descriptions of function parameters and operation.

BLAS Code Examples

Example. Using BLAS Level 1 Function

The following example illustrates a call to the BLAS Level 1 function `sdot`. This function performs a vector-vector operation of computing a scalar product of two single-precision real vectors *x* and *y*.

Parameters

n	Specifies the number of elements in vectors *x* and *y*.
incx	Specifies the increment for the elements of *x*.
incy	Specifies the increment for the elements of *y*.

```
program dot_main
real x(10), y(10), sdot, res
integer n, incx, incy, i
external sdot
n = 5
incx = 2
incy = 1
do i = 1, 10
  x(i) = 2.0e0
  y(i) = 1.0e0
end do
res = sdot (n, x, incx, y, incy)
print*, `SDOT = `, res
end
```

As a result of this program execution, the following line is printed:

SDOT = 10.000

Example. Using BLAS Level 1 Routine

The following example illustrates a call to the BLAS Level 1 routine `scopy`. This routine performs a vector-vector operation of copying a single-precision real vector *x* to a vector *y*.

Parameters

n	Specifies the number of elements in vectors *x* and *y*.
incx	Specifies the increment for the elements of *x*.
incy	Specifies the increment for the elements of *y*.

```
program copy_main
real x(10), y(10)
integer n, incx, incy, i
n = 3
incx = 3
incy = 1
do i = 1, 10
  x(i) = i
end do
call scopy (n, x, incx, y, incy)
print*, `Y = `, (y(i), i = 1, n)
end
```

As a result of this program execution, the following line is printed:

Y = 1.00000 4.00000 7.00000

Example. Using BLAS Level 2 Routine

The following example illustrates a call to the BLAS Level 2 routine `sger`. This routine performs a matrix-vector operation

```
a :=  alpha*x*y' + a.
```

Parameters

alpha	Specifies a scalar *alpha*.
x	*m*-element vector.
y	*n*-element vector.
a	*m*-by-*n* matrix.

```
program ger_main
real a(5,3), x(10), y(10), alpha
integer m, n, incx, incy, i, j, lda
m = 2
n = 3
lda = 5
incx = 2
incy = 1
alpha = 0.5
do i = 1, 10
  x(i) = 1.0
  y(i) = 1.0
end do
do i = 1, m
  do j = 1, n
    a(i,j) = j
  end do
end do
```

```
call sger (m, n, alpha, x, incx, y, incy, a, lda)
print*, `Matrix A: `
do i = 1, m
  print*, (a(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix *a* is printed as follows:

Matrix A:

1.50000 2.50000 3.50000

1.50000 2.50000 3.50000

Example. Using BLAS Level 3 Routine

The following example illustrates a call to the BLAS Level 3 routine `ssymm`. This routine performs a matrix-matrix operation

```
c :=  alpha*a*b' + beta*c.
```

Parameters

alpha	Specifies a scalar *alpha*.
beta	Specifies a scalar *beta*.
a	Symmetric matrix
b	*m*-by-*n* matrix
c	*m*-by-*n* matrix

```
program symm_main
real a(3,3), b(3,2), c(3,3), alpha, beta
integer m, n, lda, ldb, ldc, i, j
character uplo, side
uplo = 'u'
side = 'l'
m = 3
n = 2
lda = 3
ldb = 3
ldc = 3
alpha = 0.5
beta = 2.0
do i = 1, m
  do j = 1, m
    a(i,j) = 1.0
  end do
end do
do i = 1, m
  do j = 1, n
    c(i,j) = 1.0
    b(i,j) = 2.0
  end do
end do
call ssymm (side, uplo, m, n, alpha,
a, lda, b, ldb, beta, c, ldc)
print*, `Matrix C: `
```

```
do i = 1, m
  print*, (c(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix *c* is printed as follows:

Matrix C:

5.00000 5.00000

5.00000 5.00000

5.00000 5.00000

The following example illustrates a call from a C program to the Fortran version of the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

Fourier Transform Functions Code Examples

This section presents code examples for functions described in the "FFT Functions" and "Cluster FFT Functions" subsections in the "Fourier Transform Functions" section. The examples are grouped in subsections

- Examples for FFT Functions, including Examples of Using Multi-Threading for FFT Computation
- Examples for Cluster FFT Functions
- Auxiliary data transformations.

FFT Code Examples

This section presents examples of using the FFT interface functions described in "Fourier Transform Functions".

Here are the examples of two one-dimensional computations. These examples use the default settings for all of the configuration parameters, which are specified in "Configuration Settings".

In the Fortran examples, the `use mkl_dfti` statement assumes that:

- The `mkl_dfti.f90` module definition file is already compiled.
- The `mkl_dfti.mod` module file is available.

One-dimensional In-place FFT

```
! Fortran example.
! 1D complex to complex, and real to conjugate-even
Use MKL_DFTI
Complex :: X(32)
Real :: Y(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status
!...put input data into X(1),...,X(32); Y(1),...,Y(32)

! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,&
   DFTI_COMPLEX, 1, 32 )
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X )
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by {X(1),X(2),...,X(32)}

! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor(My_Desc2_Handle, DFTI_SINGLE,&
   DFTI_REAL, 1, 32)
```

```
Status = DftiCommitDescriptor(My_Desc2_Handle)
Status = DftiComputeForward(My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given in CCS format.
```

One-dimensional Out-of-place FFT

```
! Fortran example.
! 1D complex to complex, and real to conjugate-even
Use MKL_DFTI
Complex :: X_in(32)
Complex :: X_out(32)
Real :: Y_in(32)
Real :: Y_out(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status
...put input data into X_in(1),...,X_in(32); Y_in(1),...,Y_in(32)
! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,
DFTI_COMPLEX, 1, 32 )
Status = DftiSetValue( My_Desc1_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X_in, X_out )
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by {X_out(1),X_out(2),...,X_out(32)}
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor(My_Desc2_Handle, DFTI_SINGLE,
DFTI_REAL, 1, 32)
Status = DftiSetValue( My_Desc2_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor(My_Desc2_Handle)
Status = DftiComputeForward(My_Desc2_Handle, Y_in, Y_out)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given by Y_out in CCS format.
```

Two-dimensional FFT

The following is an example of two simple two-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

```
! Fortran example.
! 2D complex to complex, and real to conjugate-even
Use MKL_DFTI
Complex ::  X_2D(32,100)
Real :: Y_2D(34, 102)
Complex ::  X(3200)
Real :: Y(3468)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status, L(2)
!...put input data into X_2D(j,k), Y_2D(j,k), 1<=j=32,1<=k<=100
!...set L(1) = 32, L(2) = 100
```

```
!...the transform is a 32-by-100

! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,&
          DFTI_COMPLEX, 2, L)
Status = DftiCommitDescriptor( My_Desc1_Handle)
Status = DftiComputeForward( My_Desc1_Handle, X)
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by X_2D(j,k), 1<=j<=32, 1<=k<=100

! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc2_Handle, DFTI_SINGLE,&
          DFTI_REAL, 2, L)
Status = DftiCommitDescriptor( My_Desc2_Handle)
Status = DftiComputeForward( My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given by the complex value z(j,k) 1<=j<=32; 1<=k<=100
! and is stored in CCS format
```

The following example demonstrates how you can change the default configuration settings by using the `DftiSetValue` function.

For instance, to preserve the input data after the FFT computation, the configuration of `DFTI_PLACEMENT` should be changed to "not in place" from the default choice of "in place."

The code below illustrates how this can be done:

Changing Default Settings

```
! Fortran example
! 1D complex to complex, not in place
Use MKL_DFTI
Complex :: X_in(32), X_out(32)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status
!...put input data into X_in(j), 1<=j<=32
Status = DftiCreateDescriptor( My_Desc_Handle,& DFTI_SINGLE, DFTI_COMPLEX, 1, 32)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X_in, X_out)
Status = DftiFreeDescriptor (My_Desc_Handle)
! result is X_out(1),X_out(2),...,X_out(32)
```

Using Status Checking Functions

This example illustrates the use of status checking functions described in "Fourier Transform Functions".

```
! Fortran
type(DFTI_DESCRIPTOR), POINTER :: desc
integer status
! ...descriptor creation and other code
status =  DftiCommitDescriptor(desc)
if (status .ne. 0) then
  if (.not. DftiErrorClass(status,DFTI_NO_ERROR) then
     print *, 'Error: ', DftiErrorMessage(status)
  endif
```

```
endif
```

Computing 2D FFT by One-Dimensional Transforms

Below is an example where a 20-by-40 two-dimensional FFT is computed explicitly using one-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

```
! Fortran
use mkl_dfti
Complex ::  X_2D(20,40)
Complex ::  X(800)
Equivalence (X_2D, X)
INTEGER :: STRIDE(2)
type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim1
type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim2
! ...
Status = DftiCreateDescriptor(Desc_Handle_Dim1, DFTI_SINGLE,&
                              DFTI_COMPLEX, 1, 20 )
Status = DftiCreateDescriptor(Desc_Handle_Dim2, DFTI_SINGLE,&
                              DFTI_COMPLEX, 1, 40 )
! perform 40 one-dimensional transforms along 1st dimension
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_INPUT_DISTANCE, 20 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_OUTPUT_DISTANCE, 20 )
Status = DftiCommitDescriptor( Desc_Handle_Dim1 )
Status = DftiComputeForward( Desc_Handle_Dim1, X )
! perform 20 one-dimensional transforms along 2nd dimension
Stride(1) = 0; Stride(2) = 20
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_STRIDES, Stride )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_STRIDES, Stride )
Status = DftiCommitDescriptor( Desc_Handle_Dim2 )
Status = DftiComputeForward( Desc_Handle_Dim2, X )
Status = DftiFreeDescriptor( Desc_Handle_Dim1 )
Status = DftiFreeDescriptor( Desc_Handle_Dim2 )
```

The following code illustrates real multi-dimensional transforms with CCE format storage of conjugate-even complex matrix. Example "Two-Dimensional REAL In-place FFT (Fortran Interface)" is two-dimensional in-place transform and Example "Two-Dimensional REAL Out-of-place FFT (Fortran Interface)" is two-dimensional out-of-place transform in Fortran interface. Note that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

Two-Dimensional REAL In-place FFT

```
! Fortran example.
! 2D and real to conjugate-even
Use MKL_DFTI
```

```
Real :: X_2D(34,100) ! 34  = (32/2 + 1)*2
Real :: X(3400)
Equivalence (X_2D, X)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_in(3)
Integer :: strides_out(3)
! ...put input data into X_2D(j,k), 1<=j=32,1<=k<=100
! ...set L(1) = 32, L(2) = 100
! ...set strides_in(1) = 0, strides_in(2) = 1, strides_in(3) = 34
! ...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17
! ...the transform is a 32-by-100
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,&
DFTI_REAL, 2, L )
Status = DftiSetValue(My_Desc_Handle, DFTI_CONJUGATE_EVEN_STORAGE,&
DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue(My_Desc_Handle, DFTI_INPUT_STRIDES, strides_in)
Status = DftiSetValue(My_Desc_Handle, DFTI_OUTPUT_STRIDES, strides_out)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X )
Status = DftiFreeDescriptor(My_Desc_Handle)
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in real matrix X_2D in CCE format.
```

Two-Dimensional REAL Out-of-place FFT

```
! Fortran example.
! 2D and real to conjugate-even
Use MKL_DFTI
Real :: X_2D(32,100)
Complex :: Y_2D(17, 100) ! 17 = 32/2 + 1
Real :: X(3200)
Complex :: Y(1700)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_out(3)

! ...put input data into X_2D(j,k), 1<=j=32,1<=k<=100
! ...set L(1) = 32, L(2) = 100
! ...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17

! ...the transform is a 32-by-100
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,&
DFTI_REAL, 2, L )
Status = DftiSetValue(My_Desc_Handle,&
DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE )
Status = DftiSetValue(My_Desc_Handle,&
DFTI_OUTPUT_STRIDES, strides_out)

Status = DftiCommitDescriptor(My_Desc_Handle)
Status = DftiComputeForward(My_Desc_Handle, X, Y)
Status = DftiFreeDescriptor(My_Desc_Handle)
```

```
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in complex matrix Y_2D in CCE format.
```

Examples of Using OpenMP* Threading for FFT Computation

The following sample program shows how to employ internal OpenMP* threading in Intel® oneAPI Math Kernel Library (oneMKL) for FFT computation.

To specify the number of threads inside Intel® oneAPI Math Kernel Library (oneMKL), use the following settings:

set `MKL_NUM_THREADS = 1` for one-threaded mode;

set `MKL_NUM_THREADS = 4` for multi-threaded mode.

Using oneMKL Internal Threading Mode (C Example)

```
/* C99 example */
#include "mkl_dfti.h"

float data[200][100];
DFTI_DESCRIPTOR_HANDLE fft = NULL;
MKL_LONG dim_sizes[2] = {200, 100};

/* ...put values into data[i][j] 0<=i<=199, 0<=j<=99 */

DftiCreateDescriptor(&fft, DFTI_SINGLE, DFTI_REAL, 2, dim_sizes);
DftiCommitDescriptor(fft);
DftiComputeForward(fft, data);
DftiFreeDescriptor(&fft);
```

The following Example “Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region” and Example “Using Parallel Mode with Multiple Descriptors Initialized in One Thread” illustrate a parallel customer program with each descriptor instance used only in a single thread.

Specify the number of OpenMP threads for Example “Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region” like this:

set `MKL_NUM_THREADS = 1` for Intel® oneAPI Math Kernel Library (oneMKL) to work in the single-threaded mode (recommended);

set `OMP_NUM_THREADS = 4` for the customer program to work in the multi-threaded mode.

Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region

Note that in this example, the program can be transformed to become single-threaded at the customer level but using parallel mode within Intel® oneAPI Math Kernel Library (oneMKL). To achieve this, you must set the parameter `DFTI_NUMBER_OF_TRANSFORMS = 4` and to set the corresponding parameter `DFTI_INPUT_DISTANCE = 5000`.

```
program fft2d_private_descr_main
  use mkl_dfti

  integer nth, len(2)
! 4 OMP threads, each does 2D FFT 50x100 points
  parameter (nth = 4, len = (/50, 100/))
  complex x(len(2)*len(1), nth)

  type(dfti_descriptor), pointer :: myFFT
```

```
  integer th, myStatus

! assume x is initialized and do 2D FFTs
!$OMP PARALLEL DO SHARED(len, x) PRIVATE(myFFT, myStatus)
  do th = 1, nth
    myStatus = DftiCreateDescriptor (myFFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len)
    myStatus = DftiCommitDescriptor (myFFT)
    myStatus = DftiComputeForward (myFFT, x(:, th))
    myStatus = DftiFreeDescriptor (myFFT)
  end do
!$OMP END PARALLEL DO
end
```

Specify the number of OpenMP threads for Example "Using Parallel Mode with Multiple Descriptors Initialized in One Thread" like this:

set `MKL_NUM_THREADS` = 1 for Intel® oneAPI Math Kernel Library (oneMKL) to work in the single-threaded mode (obligatory);

set `OMP_NUM_THREADS` = 4 for the customer program to work in the multi-threaded mode.

Using Parallel Mode with Multiple Descriptors Initialized in One Thread

```
program fft2d_array_descr_main
  use mkl_dfti

  integer nth, len(2)
! 4 OMP threads, each does 2D FFT 50x100 points
  parameter (nth = 4, len = (/50, 100/))
  complex x(len(2)*len(1), nth)

  type thread_data
    type(dfti_descriptor), pointer :: FFT
  end type thread_data
  type(thread_data) :: workload(nth)

  integer th, status, myStatus

  do th = 1, nth
    status = DftiCreateDescriptor (workload(th)%FFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len)
    status = DftiCommitDescriptor (workload(th)%FFT)
  end do
! assume x is initialized and do 2D FFTs
!$OMP PARALLEL DO SHARED(len, x, workload) PRIVATE(myStatus)
  do th = 1, nth
    myStatus = DftiComputeForward (workload(th)%FFT, x(:, th))
  end do
!$OMP END PARALLEL DO
  do th = 1, nth
    status = DftiFreeDescriptor (workload(th)%FFT)
  end do
end
```

The following Example "Using Parallel Mode with a Common Descriptor" illustrates a parallel customer program with a common descriptor used in several threads.

Using Parallel Mode with a Common Descriptor

```
program fft2d_shared_descr_main
  use mkl_dfti
```

```
  integer nth, len(2)
! 4 OMP threads, each does 2D FFT 50x100 points
  parameter (nth = 4, len = (/50, 100/))
  complex x(len(2)*len(1), nth)
  type(dfti_descriptor), pointer :: FFT

  integer th, status, myStatus

  status = DftiCreateDescriptor (FFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len)
  status = DftiCommitDescriptor (FFT)
! assume x is initialized and do 2D FFTs
!$OMP PARALLEL DO SHARED(len, x, FFT) PRIVATE(myStatus)
  do th = 1, nth
    myStatus = DftiComputeForward (FFT, x(:, th))
  end do
!$OMP END PARALLEL DO
  status = DftiFreeDescriptor (FFT)
end
```

Examples for Cluster FFT Functions

The following C example computes a 2-dimensional out-of-place FFT using the cluster FFT interface:

2D Out-of-place Cluster FFT Computation

```
/* C99 example */
#include "mpi.h"
#include "mkl_cdft.h"

DFTI_DESCRIPTOR_DM_HANDLE desc = NULL;
MKL_LONG v, i, j, n, s;
Complex *in, *out;
MKL_LONG dim_sizes[2] = {nx, ny};

MPI_Init(...);

/* Create descriptor for 2D FFT */
DftiCreateDescriptorDM(MPI_COMM_WORLD,
                       &desc, DFTI_DOUBLE, DFTI_COMPLEX, 2, dim_sizes);
/* Ask necessary length of in and out arrays and allocate memory */
DftiGetValueDM(desc,CDFT_LOCAL_SIZE,&v);
in = (Complex*) malloc(v*sizeof(Complex));
out = (Complex*) malloc(v*sizeof(Complex));
/* Fill local array with initial data. Current process performs n rows,
   0 row of in corresponds to s row of virtual global array */
DftiGetValueDM(desc, CDFT_LOCAL_NX, &n);
DftiGetValueDM(desc, CDFT_LOCAL_X_START, &s);
/* Virtual global array globalIN is defined by function f as
   globalIN[i*ny+j]=f(i,j) */
for(i = 0; i < n; ++i)
   for(j = 0; j < ny; ++j) in[i*ny+j] = f(i+s,j);
/* Set that we want out-of-place transform (default is DFTI_INPLACE) */
DftiSetValueDM(desc, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
/* Commit descriptor, calculate FFT, free descriptor */
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc, in, out);
/* Virtual global array globalOUT is defined by function g as
   globalOUT[i*ny+j]=g(i,j)   Now out contains result of FFT. out[i*ny+j]=g(i+s,j) */
DftiFreeDescriptorDM(&desc);
```

```
free(in);
free(out);
MPI_Finalize();
```

1D In-place Cluster FFT Computations

The C example below illustrates one-dimensional in-place cluster FFT computations effected with a user-defined workspace:

```
/* C99 example */
#include "mpi.h"
#include "mkl_cdft.h"

DFTI_DESCRIPTOR_DM_HANDLE desc = NULL;
MKL_LONG N, v, i, n_out, s_out;
Complex *in, *work;

MPI_Init(...);
/* Create descriptor for 1D FFT */
DftiCreateDescriptorDM(MPI_COMM_WORLD, &desc, DFTI_DOUBLE, DFTI_COMPLEX, 1, N);
/* Ask necessary length of array and workspace and allocate memory */
DftiGetValueDM(desc,CDFT_LOCAL_SIZE,&v);
in = (Complex*) malloc(v*sizeof(Complex));
work = (Complex*) malloc(v*sizeof(Complex));
/* Fill local array with initial data. Local array has n elements,
   0 element of in corresponds to s element of virtual global array */
DftiGetValueDM(desc, CDFT_LOCAL_NX, &n);
DftiGetValueDM(desc, CDFT_LOCAL_X_START, &s);
/* Set work array as a workspace */
DftiSetValueDM(desc, CDFT_WORKSPACE, work);
/* Virtual global array globalIN is defined by function f as globalIN[i]=f(i) */
for(i = 0; i < n; ++i) in[i] = f(i+s);
/* Commit descriptor, calculate FFT, free descriptor */
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc,in);
DftiGetValueDM(desc, CDFT_LOCAL_OUT_NX, &n_out);
DftiGetValueDM(desc, CDFT_LOCAL_OUT_X_START, &s_out);
/* Virtual global array globalOUT is defined by function g as globalOUT[i]=g(i)
   Now in contains result of FFT. Local array has n_out elements,
   0 element of in corresponds to s_out element of virtual global array.
   in[i]==g(i+s_out) */
DftiFreeDescriptorDM(&desc);
free(in);
free(work);
MPI_Finalize();
```

Auxiliary Data Transformations

This section presents C examples for conversion from the Cartesian to polar representation of complex data and vice versa.

Conversion from Cartesian to polar representation of complex data

```
// Cartesian->polar conversion of complex data
// Cartesian representation: z = re + I*im
// Polar representation: z = r * exp( I*phi )
```

```
#include <mkl_vml.h>

void
variant1_Cartesian2Polar(int n,const double *re,const double *im,
                         double *r,double *phi)
{
    vdHypot(n,re,im,r);         // compute radii r[]
    vdAtan2(n,im,re,phi);       // compute phases phi[]
}

void
variant2_Cartesian2Polar(int n,const MKL_Complex16 *z,double *r,double *phi,
                         double *temp_re,double *temp_im)
{
    vzAbs(n,z,r);               // compute radii r[]
    vdPackI(n, (double*)z + 0, 2, temp_re);
    vdPackI(n, (double*)z + 1, 2, temp_im);
    vdAtan2(n,temp_im,temp_re,phi); // compute phases phi[]
}
```

Conversion from polar to Cartesian representation of complex data

```
// Polar->Cartesian conversion of complex data.
// Polar representation: z = r * exp( I*phi )
// Cartesian representation: z = re + I*im
#include <mkl_vml.h>

void
variant1_Polar2Cartesian(int n,const double *r,const double *phi,
                         double *re,double *im)
{
    vdSinCos(n,phi,im,re);      // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,re,re);           // scale real part
    vdMul(n,r,im,im);           // scale imaginary part
}

void
variant2_Polar2Cartesian(int n,const double *r,const double *phi,
                         MKL_Complex16 *z,
                         double *temp_re,double *temp_im)
{
    vdSinCos(n,phi,temp_im,temp_re); // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,temp_im,temp_im); // scale imaginary part
    vdMul(n,r,temp_re,temp_re); // scale real part
    vdUnpackI(n,temp_re,(double*)z + 0, 2); // fill in result.re
    vdUnpackI(n,temp_im,(double*)z + 1, 2); // fill in result.im
}
```

Appendix F: oneMKL Functionality

This appendix provides an overview of the Intel® oneAPI Math Kernel Library (oneMKL) functionality on the different devices.

BLAS Functionality

Fortran

Functionality	CPU	OpenMP Offload Intel GPU
Level 1 BLAS (standard)	All	All
Level 2 BLAS (standard)	All	All
Level 3 BLAS (standard)	All	All
BLAS Extensions and Specializations	{AXPY,GEMM,TRSM}_BATCH (group and strided)	{AXPY,GEMM,TRSM}_BATCH (strided)
	GEMMT, AXPBY, GEMM3M	GEMMT
	Integer GEMM (s8u8)	N/A
	JIT GEMM API	N/A
	PACK GEMM API	N/A
	COMPACT GEMM API	N/A

Transposition Functionality

Functionality	CPU	OpenMP Offload Intel GPU
In-place dense matrix transpose	Yes	No
Out-of-place dense matrix transpose	Yes	No
In-place dense matrix add	Yes	No
Out-of-place dense matrix copy	Yes	No

LAPACK Functionality

NOTE All of the DPC++ LAPACK computational routines have a corresponding `*_scratchpad_size` function for calculating the required amount of scratchpad space.

LU Factorization Routines

Functionality	CPU	OpenMP Offload Intel GPU
getrf	Yes	Yes
getrs	Yes	Yes
getri	Yes	Yes

Cholesky Factorization Routines

Functionality	CPU	OpenMP Offload Intel GPU
potrf	Yes	Yes
potrs	Yes	Yes
potri	Yes	Yes

Orthogonal Factorization Routines

Functionality	CPU	OpenMP Offload Intel GPU
geqrf	Yes	Yes
{or,un}gqr	Yes	Yes
{or,un}mqr	Yes	Yes
gerqf	Yes	No
{or,un}mrq	Yes	No

Other Linear Equation Routines

Functionality	CPU	OpenMP Offload Intel GPU
trtrs	Yes	Yes
{sy,he}trf	Yes	No

Symmetric Eigenvalue Routines

Functionality	CPU	OpenMP Offload Intel GPU
{sy,he}ev	Yes	Yes
{sy,he}evd	Yes	Yes
{sy,he}evx	Yes	Yes
{sy,he}trd	Yes	Yes
{or,un}gtr	Yes	No
{or,un}mtr	Yes	No
steqr	Yes	Yes

Generalized Symmetric Eigenvalue Routines

Functionality	CPU	OpenMP Offload Intel GPU
{sy,he}gvd	Yes	Yes
{sy,he}gvx	Yes	Yes

Singular Value Routines

Functionality	CPU	OpenMP Offload Intel GPU
gesvd	Yes	Yes
gebrd	Yes	Yes
{or,un}gbr	Yes	No

Batched LAPACK Routines

Functionality	CPU	OpenMP Offload Intel GPU (ILP64 Interface)
getrf_batch	Strided	Strided
getrfnp_batch	Strided	Strided
getrs_batch	Strided	Strided
getrsnp_batch	Strided	Strided
getri_batch	No	No

Functionality	CPU	OpenMP Offload Intel GPU (ILP64 Interface)
potrf_batch	No	No
potrs_batch	No	No
geqrf_batch	No	No
{or,un}gqr_batch	No	No

Other LAPACK Routines

CPU	OpenMP Offload Intel GPU
Yes	No

DFT Functionality

DFTI Interfaces

Functionality	CPU	OpenMP Offload Intel GPU
Complex-to-Complex FFT transformations	Yes, 1D through 7D	Yes, 1D through 3D
Real-to-Complex FFT transformations	Yes, 1D through 7D	Yes, 1D through 3D

FFTW3 Interfaces

Functionality	CPU	OpenMP Offload Intel GPU
Complex-to-Complex FFT transformations	Yes, 1D through 7D	Yes, 1D through 3D
Real-to-Complex FFT transformations	Yes, 1D through 7D	No

FFTW2 Interfaces

Functionality	CPU	OpenMP Offload Intel GPU
Complex-to-Complex FFT transformations	Yes, 1D through 7D	No
Real-to-Complex FFT transformations	Yes, 1D through 7D	No

Sparse BLAS Functionality

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w,v, dense matrices = X,Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Level 1

Functionality	Operations	CPU	OpenMP Offload Intel GPU
Sparse Vector - Dense Vector addition (AXPY)	y <- alpha * w + y	Yes	No
Sparse Vector - Sparse Vector Dot product (SPDOT) (sv.sv -> sc)	d <- dot(w,v)	N/A	N/A
	dot(w,v) = sum(w_i* v_i)	No	No

Functionality	Operations	CPU	OpenMP Offload Intel GPU
	dot(w,v) = sum(conj(w_i) * v_i)	No	No
Sparse Vector - Dense Vector Dot product (SPDOT) (sv.dv -> sc)	d <- dot(w,x)	N/A	N/A
	dot(w,v) = sum(w_i* v_i)	Yes	No
	dot(w,v) = sum(conj(w_i) * v_i)	Yes	No
Dense Vector - Sparse Vector Conversion (sv <-> dv)	—	N/A	N/A
	x = scatter(w)	Yes	No
	w = gather(x,windx)	Yes	No

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w,v, dense matrices = X,Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Level 2

Functionality	Operations	CPU	OpenMP Offload Intel GPU
General Matrix-Vector multiplication (GEMV) (sm*dv->dv)	y <- beta*y + alpha * op(A)*x	N/A	N/A
	op(A) = A	Yes	No
	op(A) = A^T	Yes	No
	op(A) = A^H	Yes	No
Symmetric Matrix-Vector multiplication (SYMV) (sm*dv->dv)	y <- beta*y + alpha * op(A)*x	N/A	N/A
	op(A) = A	Yes	No
	op(A) = A^T	No	No
	op(A) = A^H	Yes	No
Triangular Matrix-Vector multiplication (TRMV) (sm*dv->dv)	y <- beta*y + alpha * op(A)*x	N/A	N/A
	op(A) = A	Yes	No
	op(A) = A^T	Yes	No
	op(A) = A^H	Yes	No
General Matrix-Vector mult with dot product (GEMVDOT) (sm*dv -> dv, dv.dv->sc)	y <- beta*y + alpha * op(A)*x, d = dot(x,y)	N/A	N/A
	op(A) = A	Yes	No
	op(A) = A^T	Yes	No
	op(A) = A^H	Yes	No
Triangular Solve (TRSV) (inv(sm)*dv -> dv)	solve for y, op(A)*y = alpha*x	N/A	N/A
	op(A) = A	Yes	No
	op(A) = A^T	Yes	No
	op(A) = A^H	Yes	No

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w,v, dense matrices = X,Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Level 3

Functionality	Operations	CPU	OpenMP Offload Intel GPU
General Sparse Matrix - Dense Matrix Multiplication (GEMM) (sm*dm->dm)	Y <- alpha*op(A)*op(X) + beta*Y	N/A	N/A
	op(A) = A, op(X) = X	Yes	No
	op(A) = A^T, op(X) = X	Yes	No
	op(A) = A^H, op(X) = X	Yes	No
	op(A) = A, op(X) = X^T	No	No
	op(A) = A^T, op(X) = X^T	No	No
	op(A) = A, op(X) = X^H	No	No
	op(A) = A^H	No	No
	op(A) = A^T, op(X) = X^H	No	No
	op(A) = A^H, op(X) = X^H	No	No
General Dense Matrix - Sparse Matrix Multiplication (GEMM) (dm*sm->dm)	Y <- alpha*op(X)*op(A) + beta*Y	N/A	N/A
	op(X) = X, op(A)=A	No	No
	op(X) = X^H, op(A)=A	No	No
	op(X) = X^H, op(A)=A	No	No
	op(X) = X, op(A)=A^H	No	No
	op(X) = X^H, op(A)=A^H	No	No
	op(X) = X^H, op(A)=A^H	No	No
	op(X) = X, op(A)=A^H	No	No
	op(X) = X^H, op(A)=A^H	No	No
	op(X) = X^H, op(A)=A^H	No	No
General Sparse Matrix - Sparse Matrix Multiplication (GEMM) (sm*sm->sm)	C <- alpha*op(A)*op(B) + beta*C	N/A	N/A
	op(A)=A, op(B)=B	Yes	No
	op(A)=A^T, op(B)=B	Yes	No
	op(A)=A^H, op(B)=B	Yes	No
	op(A)=A, op(B)=B^T	Yes	No
	op(A)=A^T, op(B)=B^T	Yes	No
	op(A)=A^H, op(B)=B^T	Yes	No
	op(A)=A, op(B)=B^H	Yes	No

Functionality	Operations	CPU	OpenMP Offload Intel GPU
	op(A)=A^T, op(B)=B^H	Yes	No
	op(A)=A^H, op(B)=B^H	Yes	No
General Sparse Matrix - Sparse Matrix Multiplication (GEMM) (sm*sm->dm)	Y <- alpha*op(A)*op(B) + beta*Y	N/A	N/A
	op(A)=A, op(B)=B	Yes	No
	op(A)=A^T, op(B)=B	Yes	No
	op(A)=A^H, op(B)=B	Yes	No
	op(A)=A, op(B)=B^T	No	No
	op(A)=A^T, op(B)=B^T	No	No
	op(A)=A^H, op(B)=B^T	No	No
	op(A)=A, op(B)=B^H	No	No
	op(A)=A^T, op(B)=B^H	No	No
	op(A)=A^H, op(B)=B^H	No	No
Symmetric Rank-K update (SYRK) (sm*sm->sm)	C <- op(A)*op(A)^H	N/A	N/A
	op(A)=A	Yes	No
	op(A)=A^T	Yes	No
	op(A)=A^H	Yes	No
Symmetric Rank-K update (SYRK) (sm*sm->dm)	Y <- op(A)*op(A)^H	N/A	N/A
	op(A)=A	Yes	No
	op(A)=A^T	Yes	No
	op(A)=A^H	Yes	No
Symmetric Triple Product (SYPR) (op(sm)*sm*sm -> sm)	C <- op(A)*B*op(A)^H	N/A	N/A
	op(A)=A	Yes	No
	op(A)=A^T	Yes	No
	op(A)=A^H	Yes	No
Triangular Solve (TRSM) (inv(sm)*dm -> dm)	solve for Y, op(A)*Y = alpha*X	N/A	N/A
	op(A)=A	Yes	No
	op(A)=A^T	Yes	No
	op(A)=A^H	Yes	No

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w,v, dense matrices = X,Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Other

Functionality	Operations	CPU	OpenMP Offload Intel GPU
Symmetric Gauss-Seidel Preconditioner (SYMGS) (update A*x=b, A=L+D+U)	x0 <- x*alpha; (L +D)*x1=b-U*x0; (U +D)*x=b-L*x1	Yes	No
Symmetric Gauss-Seidel Preconditioner with Matrix-Vector product (SYMGS_MV) (update A*x=b, A=L+D+U)	x0 <- x*alpha; (L +D)*x1=b-U*x0; (U +D)*x=b-L*x1; y=A*x	Yes	No
LU Smoother (LU_SMOOTHER) (update A*x=b, A=L+D+U, E~inv(D))	r=b-A*x; (L+D)*E*(U +D)*dx=r; y=x+dr	Yes	No
Sparse Matrix Add (ADD)	C <- alpha*op(A) + B	Yes	No
	op(A) = A^T	Yes	No
	op(A) = A^H	Yes	No

In the following table for operations, dense vectors = x, y, sparse vectors = w,v, dense matrices = X,Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Helper Functions

Functionality	Operations	CPU	OpenMP Offload Intel GPU
Sort Indices of Matrix (ORDER)	N/A	Yes	No
Transpose of Sparse Matrix (TRANSPOSE)	A <- op(A) with op=trans or conjtrans	N/A	N/A
	transpose CSR/CSC matrix	Yes	No
	transpose BSR matrix	Yes	No
Sparse Matrix Format Converter (CONVERT)	N/A	Yes	No
Dense to Sparse Matrix Format Converter (CONVERT)	N/A	Yes	No
Copy Matrix Handle (COPY)	N/A	Yes	No
Create CSR Matrix Handle	N/A	Yes	No
Create CSC Matrix Handle	N/A	Yes	No
Create COO Matrix Handle	N/A	Yes	No
Create BSR Matrix Handle	N/A	Yes	No
Export CSR Matrix	Allows access to internal data in the CSR Matrix handle	Yes	No
Export CSC Matrix	Allows access to internal data in the CSC Matrix handle	Yes	No
Export COO Matrix	Allows access to internal data in the COO Matrix handle	Yes	No

Functionality	Operations	CPU	OpenMP Offload Intel GPU
Export BSR Matrix	Allows access to internal data in the BSR Matrix handle	Yes	No
Set Value in Matrix	N/A	Yes	No

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w,v, dense matrices = X,Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Optimize Stages

Functionality	Operations	CPU	OpenMP Offload Intel GPU
add MEMORY hint and optimize	Chooses to allow larger memory requiring optimizations or not.	Yes	No
Add GEMV hint and optimize	N/A	Yes	No
Add SYMV hint and optimize	N/A	Yes	No
Add TRMV hint and optimize	N/A	Yes	No
add TRSV hint and optimize	N/A	Yes	No
add GEMM hint and optimize	N/A	Yes	No
add TRSM hint and optimize	N/A	Yes	No
add DOTMV hint and optimize	N/A	Yes	No
add SYMGS hint and optimize	N/A	Yes	No
add SYMGS_MV hint and optimize	N/A	Yes	No
add LU_SMOOTHER hint and optimize	N/A	Yes	No

Sparse Solvers Functionality

Functionality	CPU	OpenMP Offload Intel GPU
Sparse Cholesky Factorization	Yes	No
Sparse LU Factorization	Yes	No
Sparse QR factorization	Yes	No
Hermitian/Symmetric Eigensolver on intervals for Sparse Matrices	Yes	No
Extremal Eigensolvers for Sparse Matrix	Yes	No
Poisson Solver	Yes	No
Trust Region Solver	Yes	No

Random Number Generators Functionality

Engines

Functionality	CPU	OpenMP Offload Intel GPU
MRG32K3A	Yes	Yes
MT2203	Yes	Yes
MT19937	Yes	Yes
PHILOX4X32X10	Yes	Yes
SOBOL	Yes	Yes
ARS5	Yes	No
MCG59	Yes	Yes
NIEDERR	Yes	No
MCG31	Yes	Yes
WH	Yes	No
SFMT19937	Yes	No
R250	Yes	No
NONDETERM	Yes	No
DABSTRACT	No	No
SABSTRACT	No	No
IABSTRACT	No	No

Distributions

Functionality	CPU	OpenMP Offload Intel GPU
Uniform (single/double/integer)	Yes	Yes
UniformBits32	Yes	Yes
UniformBits64		
Lognormal (single/double)	Yes	Yes
Gaussian (single/double)	Yes	Yes
Poisson	Yes	Yes
UniformBits	Yes	Yes
Bernoulli	Yes	Yes
Beta (single/double)	Yes	No
Binomial	Yes	No
ChiSquare (single/double)	Yes	No
Exponential (single/double)	Yes	Yes
Gamma (single/double)	Yes	No

Functionality	CPU	OpenMP Offload Intel GPU
Geometric	Yes	Yes
Gumbel (single/double)	Yes	No
Hyper Geometric	Yes	No
Laplace (single/double)	Yes	Yes
Multinomial	Yes	No
Negative Binomial	Yes	No
PoissonV	Yes	No
Rayleigh (single/double)	Yes	Yes
Weibull (single/double)	Yes	Yes
Cauchy (single/double)	Yes	Yes
GaussianMV (single/double)	Yes	Yes

Vector Math Functionality

Functionality	CPU	OpenMP Offload Intel GPU
Vector Math Functions, Single Precision	Yes	Yes*
Vector Math Functions, Double Precision	Yes	Yes*
Vector Math Functions, Single Precision Complex	Yes	No
Vector Math Functions, Double Precision Complex	Yes	No

OpenMP offload to the GPU is implemented in the Linux OS, but not in the Windows* OS. The Windows OS implementation will be available in a future release.

Data Fitting Functionality

Splines, Spline Type

Functionality	CPU	OpenMP Offload Intel GPU
default (linear/quadratic/cubic)	Yes	No
subbotin	Yes	No
natural	Yes	No
hermite	Yes	No
akima	Yes	No
bessel	Yes	No
hyman	Yes	No
lookup interpolant	Yes	No
cr stepwise const interpolant	Yes	No

Functionality	CPU	OpenMP Offload Intel GPU
cl stepwise const interpolant	Yes	No

Computation Routines

Functionality	CPU	OpenMP Offload Intel GPU
Construct1D	Yes	No
Interpolate1D/Interpolate1DEx	Yes	No
Integrate1D/Integrate1DEx	Yes	No
SearchCells1D/SearchCells1DEx	Yes	No
InterpolationCallBack	Yes	No
IntegrateCallBack	Yes	No
SearchCellsCallBack	Yes	No

Summary Statistics Functionality

Functionality	CPU	OpenMP Offload Intel GPU
min	Yes	No
max	Yes	No
raw sum	Yes	No
2nd order raw sum		
3rd order raw sum		
4th order raw sum		
2nd order central sum	Yes	No
3rd order central sum		
4th order central sum		
mean	Yes	No
2nd order raw moment		
3rd order raw moment		
4th order raw moment		
2nd order central moment	Yes	No
3rd order central moment		
4th order central moment		
kurtosis	Yes	No
skewness	Yes	No
variation coefficient	Yes	No
covariance matrix	Yes	No
correlation matrix	Yes	No
cross-product matrix	Yes	No

Functionality	CPU	OpenMP Offload Intel GPU
pooled covariance matrix	Yes	No
pooled mean	Yes	No
group covariance matrix	Yes	No
group mean	Yes	No
quantiles	Yes	No
order statistics	Yes	No
robust covariance matrix	Yes	No
ouliers detection	Yes	No
partial covariance matrix	Yes	No
partial covariance matrix	Yes	No
missing values	Yes	No
parameterized correlation matrix	Yes	No
stream quantiles	Yes	No
mean absolute deviation	Yes	No
median absolute deviation	Yes	No
sorted observations	Yes	No

Bibliography

For more information about the BLAS, Sparse BLAS, LAPACK, ScaLAPACK, Sparse Solver, Extended Eigensolver, VM, VS, FFT, and Non-Linear Optimization Solvers functionality, refer to the following publications:

- **BLAS Level 1**

  C. Lawson, R. Hanson, D. Kincaid, and F. Krough. *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, Vol.5, No.3 (September 1979) 308-325.
- **BLAS Level 2**

  J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.14, No.1 (March 1988) 1-32.
- **BLAS Level 3**

  J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software (December 1989).
- **Sparse BLAS**

  D. Dodson, R. Grimes, and J. Lewis. *Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Math Software, Vol.17, No.2 (June 1991).

  D. Dodson, R. Grimes, and J. Lewis. *Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.17, No.2 (June 1991).

[Duff86] I.S.Duff, A.M.Erisman, and J.K.Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986.

[CXML01] *Compaq Extended Math Library*. Reference Guide, Oct.2001.

[Rem05] K.Remington. *A NIST FORTRAN Sparse Blas User's Guide*. (available on http://math.nist.gov/~KRemington/fspblas/)

[Saad94] Y.Saad. SPARSKIT: *A Basic Tool-kit for Sparse Matrix Computation*. Version 2, 1994.(http://www.cs.umn.edu/~saad)

[Saad96] Y.Saad. *Iterative Methods for Linear Systems*. PWS Publishing, Boston, 1996.

- **LAPACK**

[AndaPark94] A. A. Anda and H. Park. *Fast plane rotations with dynamic scaling*, SIAM J. matrix Anal. Appl., Vol. 15 (1994), pp. 162-174.

[Baudin12] M. Baudin, R. Smith. *A Robust Complex Division in Scilab*, available from http://www.arxiv.org, arXiv:1210.4539v2 (2012).

[Bischof00] C. H. Bischof, B. Lang, and X. Sun. *Algorithm 807: The SBR toolbox-software for successive band reduction*, ACM Transactions on Mathematical Software, Vol. 26, No. 4, pages 602-616, December 2000.

[Demmel92] J. Demmel and K. Veselic. *Jacobi's method is more accurate than QR*, SIAM J. Matrix Anal. Appl. 13(1992):1204-1246.

[Demmel12] J. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou. *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM Journal on Scientific Computing, Vol. 34, No 1, 2012.

[deRijk98] P. P. M. De Rijk. *A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer*, SIAM J. Sci. Stat. Comp., Vol. 10 (1998), pp. 359-371.

[Dhillon04] I. Dhillon, B. Parlett. *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Algebra and its Applications, 387(1), pp. 1-28, August 2004.

[Dhillon04-02] I. Dhillon, B. Parlett. *Orthogonal Eigenvectors and * Relative Gaps*, SIAM Journal on Matrix Analysis and Applications, Vol. 25, 2004. (Also LAPACK Working Note 154.)

[Dhillon97] I. Dhillon. *A new O(n^2) algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem*, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.

[Drmac08-1] Z. Drmac and K. Veselic. *New fast and accurate Jacobi SVD algorithm I*, SIAM J. Matrix Anal. Appl. Vol. 35, No. 2 (2008), pp. 1322-1342. LAPACK Working note 169.

[Drmac08-2] Z. Drmac and K. Veselic. *New fast and accurate Jacobi SVD algorithm II*, SIAM J. Matrix Anal. Appl. Vol. 35, No. 2 (2008), pp. 1343-1362. LAPACK Working note 170.

[Drmac08-3] Z. Drmac and K. Bujanovic. *On the failure of rank-revealing QR factorization software - a case study*, ACM Trans. Math. Softw. Vol. 35, No 2 (2008), pp. 1-28. LAPACK Working note 176.

[Drmac08-4] Z. Drmac. *Implementation of Jacobi rotations for accurate singular value computation in floating point arithmetic*, SIAM J. Sci. Comp., Vol. 18 (1997), pp. 1200-1222.

[Elmroth00] E. Elmroth and F. Gustavson. *Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance*, IBM J. Research & Development, Vol. 44, No. 4, 2000, pp 605-624.

[Golub96] G. Golub and C. Van Loan. *Matrix Computations*, Johns Hopkins University Press, Baltimore, third edition,1996.

[LUG] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*, Third Edition, Society for Industrial and Applied Mathematics (SIAM), 1999.

[Kahan66] W. Kahan. *Accurate Eigenvalues of a Symmetric Tridiagonal Matrix*, Report CS41, Computer Science Dept., Stanford University, July 21, 1966.

[Marques06] O.Marques, E.J.Riedy, and Ch.Voemel. *Benefits of IEEE-754 Features in Modern Symmetric Tridiagonal Eigensolvers*, SIAM Journal on Scientific Computing, Vol.28, No.5, 2006. (Tech report version in LAPACK Working Note 172 with the same title.)

[Sutton09] Brian D. Sutton. *Computing the complete CS decomposition*, Numer. Algorithms, 50(1):33-65, 2009.

- **ScaLAPACK**

[SLUG] L. Blackford, J. Choi, A.Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K.Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics (SIAM), 1997.

- **Sparse Solver**

[Duff99] I. S. Duff and J. Koster. *The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices.* SIAM J. Matrix Analysis and Applications, 20(4):889-901, 1999.

[Dong95] J. Dongarra, V.Eijkhout, A.Kalhan. *Reverse Communication Interface for Linear Algebra Templates for Iterative Methods*. UT-CS-95-291, May 1995. http://www.netlib.org/lapack/lawnspdf/lawn99.pdf

[Li99] X.S. Li and J.W. Demmel. *A Scalable Sparse Direct Solver Using Static Pivoting*. In Proceeding of the 9th SIAM conference on Parallel Processing for Scientific Computing, San Antonio, Texas, March 22-34,1999.

[Liu85] J.W.H. Liu. *Modification of the Minimum-Degree algorithm by multiple elimination*. ACM Transactions on Mathematical Software, 11(2):141-153, 1985.

[Menon98] R. Menon L. Dagnum. *OpenMP: An Industry-Standard API for Shared-Memory Programming*. IEEE Computational Science & Engineering, 1:46-55, 1998. http://www.openmp.org.

[Saad03] Y. Saad. *Iterative Methods for Sparse Linear Systems.* 2nd edition, SIAM, Philadelphia, PA, 2003.

[Schenk00] O. Schenk. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*. PhD thesis, ETH Zurich, 2000.

[Schenk00-2] O. Schenk, K. Gartner, and W. Fichtner. *Efficient Sparse LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors*. BIT, 40(1):158-176, 2000.

[Schenk01] O. Schenk and K. Gartner. *Sparse Factorization with Two-Level Scheduling in PARDISO*. In Proceeding of the 10th SIAM conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, March 12-14, 2001.

[Schenk02] O. Schenk and K. Gartner. *Two-level scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems*. Parallel Computing, 28:187-197, 2002.

[Schenk03] O. Schenk and K. Gartner. *Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO*. Journal of Future Generation Computer Systems, 20(3):475-487, 2004.

[Schenk04] O. Schenk and K. Gartner. *On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems*. Technical Report, Department of Computer Science, University of Basel, 2004, submitted.

[Sonn89] P. Sonneveld. *CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems*. SIAM Journal on Scientific and Statistical Computing, 10:36-52, 1989.

[Young71] D.M.Young. *Iterative Solution of Large Linear Systems*. New York, Academic Press, Inc., 1971.

- **Extended Eigensolver**

[Polizzi09] E. Polizzi, *Density-Matrix-Based Algorithms for Solving Eigenvalue Problems*, Phys. Rev. B. Vol. 79, 115112, 2009.

[Polizzi12] E. Polizzi, *A High-Performance Numerical Library for Solving Eigenvalue Problems: FEAST Solver v2.0 User's Guide*, arxiv.org/abs/1203.4031, 2012.

[Bai00] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe and H. van der Vorst, editors, *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, 2000.

[Sleijpen96] G. L. G. Sleijpen and H. A. van der Vorst. *A Jacobi-Davidson iteration method for linear eigenvalue problems*. SIAM J. Matrix Anal. Appl., 17:401-425, 1996.

- **VS**

[AVX] Intel. *Intel® Advanced Vector Extensions Programming Reference*.

[Billor00] Nedret Billor, Ali S. Hadib, and Paul F. Velleman. *BACON: blocked adaptive computationally efficient outlier nominators*. Computational Statistics & Data Analysis, 34, 279-298, 2000.

[Bratley87] Bratley P., Fox B.L., and Schrage L.E. *A Guide to Simulation*. 2nd edition. Springer-Verlag, New York, 1987.

[Bratley88] Bratley P. and Fox B.L. *Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.

[Bratley92] Bratley P., Fox B.L., and Niederreiter H. *Implementation and Tests of Low-Discrepancy Sequences*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.

[BMT] Intel. *Bull Mountain Technology Software Implementation Guide*.

[Coddington94] Coddington, P. D. *Analysis of Random Number Generators Using Monte Carlo Simulation*. Int. J. Mod. Phys. C-5, 547, 1994.

[Fritsch80] Fritsch, F. N and Carlson, R. E. *Monotone Piecewise Cubic Interpolation*. SIAM Journal on Numerical Analysis (SIAM) 17 (2): 238-246, 1980.

[Gentle98] Gentle, James E. *Random Number Generation and Monte Carlo Methods*, Springer-Verlag New York, Inc., 1998.

[Hyman83] Hyman, J. M. *Accurate monotonicity preserving cubic interpolation*, SIAM J. Sci. Stat. Comput. 4, 645-654, 1983.

[IntelSWMan] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 3 vols.

[L'Ecuyer94] L'Ecuyer, Pierre. *Uniform Random Number Generation*. Annals of Operations Research, 53, 77-120, 1994.

[L'Ecuyer99] L'Ecuyer, Pierre. *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. Mathematics of Computation, 68, 225, 249-260, 1999.

[L'Ecuyer99a] L'Ecuyer, Pierre. *Good Parameter Sets for Combined Multiple Recursive Random Number Generators*. Operations Research, 47, 1, 159-164, 1999.

[L'Ecuyer01] L'Ecuyer, Pierre. *Software for Uniform Random Number Generation: Distinguishing the Good and the Bad*. Proceedings of the 2001 Winter Simulation Conference, IEEE Press, 95-105, Dec. 2001.

[Kirkpatrick81] Kirkpatrick, S., and Stoll, E. *A Very Fast Shift-Register Sequence Random Number Generator*. Journal of Computational Physics, V. 40. 517-526, 1981.

[Knuth81] Knuth, Donald E. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[Maronna02] Maronna, R.A., and Zamar, R.H., *Robust Multivariate Estimates for High-Dimensional Datasets*, Technometrics, 44, 307-317, 2002.

[Matsumoto98] Matsumoto, M., and Nishimura, T. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.

[Matsumoto00] Matsumoto, M., and Nishimura, T. *Dynamic Creation of Pseudorandom Number Generators*, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html.

[NAG] NAG Numerical Libraries. http://www.nag.co.uk/numeric/numerical_libraries.asp

[Rocke96] David M. Rocke, *Robustness properties of S-estimators of multivariate location and shape in high dimension*. The Annals of Statistics, 24(3), 1327-1345, 1996.

[Saito08] Saito, M., and Matsumoto, M. *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*. Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, Pages 607 – 622, 2008.

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html

[Salmon11] Salmon, John K., Morales, Mark A., Dror, Ron O., and Shaw, David E., *Parallel Random Numbers: As Easy as 1, 2, 3*. SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.

[Schafer97] Schafer, J.L., *Analysis of Incomplete Multivariate Data*. Chapman & Hall, 1997.

[Sobol76] Sobol, I.M., and Levitan, Yu.L. *The production of points uniformly distributed in a multidimensional cube*. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).

[SSL Notes] *Intel® oneMKL Summary Statistics Application Notes*, a document present on the Intel® oneMKL product at https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html

[VS Notes] *Intel® oneMKL Vector Statistics Notes*, a document present on the Intel® oneMKL product at https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html

[VS Data] *Intel® oneMKL Vector Statistics Performance*, a document present on the Intel® oneMKL product at https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html

- **VM**

[C99] ISO/IEC 9899:1999/Cor 3:2007. Programming languages -- C.

[Muller97] J.M.Muller. *Elementary functions: algorithms and implementation*, Birkhauser Boston, 1997.

[IEEE754] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-2008.

[VM Data] *Intel® oneMKL Vector Mathematics Performance and Accuracy*, a document present on the Intel® oneMKL product at https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html

- **FFT**

[1] E. Oran Brigham, *The Fast Fourier Transform and Its Applications*, Prentice Hall, New Jersey, 1988.

[2] Athanasios Papoulis, *The Fourier Integral and its Applications*, 2nd edition, McGraw-Hill, New York, 1984.

[3] Ping Tak Peter Tang, *DFTI - a new interface for Fast Fourier Transform libraries*, ACM Transactions on Mathematical Software, Vol. 31, Issue 4, Pages 475 - 507, 2005.

[4] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.

- **Optimization Solvers**

[Conn00] A. R. Conn, N. I.M. Gould, P. L. Toint.*Trust-region Methods.*SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, MPS-SIAM Series on Optimization edition, 2000.

- **Data Fitting Functions**

[deBoor2001] Carl deBoor. *A Practical Guide to Splines.* Revised Edition. Springer-Verlag New York Berlin Heidelberg, 2001.

[Schumaker2007] Larry L Schumaker. *Spline Functions: Basic Theory.* 3rd Edition. Cambridge University Press, Cambridge, 2007.

[StechSub76] S.B. Stechhkin, and Yu Subbotin. *Splines in Numerical Mathematics.* Izd. Nauka, Moscow, 1976.

For a reference implementation of BLAS, sparse BLAS, LAPACK, and ScaLAPACK packages visit www.netlib.org.

Glossary

A^H	Denotes the conjugate transpose of a general matrix *A*. *See also* conjugate matrix.
A^T	Denotes the transpose of a general matrix *A*. *See also* transpose.

band matrix
: A general *m*-by-*n* matrix *A* such that $a_{ij} = 0$ for $|i - j| > l$, where $1 < l < \min(m, n)$. For example, any tridiagonal matrix is a band matrix.

band storage
: A special storage scheme for band matrices. A matrix is stored in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

BLAS
: Abbreviation for Basic Linear Algebra Subprograms. These subprograms implement vector, matrix-vector, and matrix-matrix operations.

BRNG
: Abbreviation for Basic Random Number Generator. Basic random number generators are pseudorandom number generators imitating i.i.d. random number sequences of uniform distribution. Distributions other than uniform are generated by applying different transformation techniques to the sequences of random numbers of uniform distribution.

BRNG registration
: Standardized mechanism that allows a user to include a user-designed BRNG into the VSL and use it along with the predefined VSL basic generators.

Bunch-Kaufman factorization
: Representation of a real symmetric or complex Hermitian matrix *A* in the form $A = PUDU^{H}P^{T}$ (or $A = PLDL^{H}P^{T}$) where *P* is a permutation matrix, *U* and *L* are upper and lower triangular matrices with unit diagonal, and *D* is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. *U* and *L* have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of *D*.

c
: When found as the first letter of routine names, `c` indicates the use of the single-precision complex data type.

CBLAS
: C interface to the BLAS. *See* BLAS.

CDF
: Cumulative Distribution Function. The function that determines probability distribution for univariate or multivariate random variable *X*. For univariate distribution the cumulative distribution function is the function of real argument *x*, which for every *x* takes a value equal to probability of the event $A: X \le x$. For multivariate distribution the cumulative distribution function is the function of a real vector $x = (x_1, x_2, \ldots, x_n)$, which, for every *x*, takes a value equal to probability of the event $A = (X_1 \le x_1 \ \& \ X_2 \le x_2, \ \& \ \ldots, \ \& \ X_n \le x_n)$.

Cholesky factorization
: Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix *A* in the form $A = U^{H}U$ or $A = LL^{H}$, where *L* is a lower triangular matrix and *U* is an upper triangular matrix.

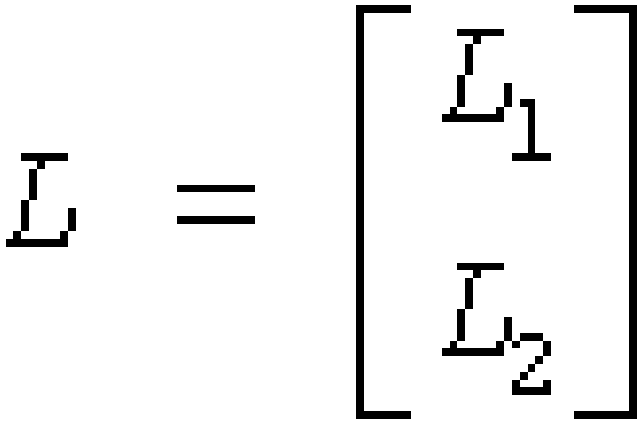
condition number
: The number $\kappa(A)$ defined for a given square matrix *A* as follows: $\kappa(A) = ||A|| \ ||A^{-1}||$.

conjugate matrix
: The matrix A^{H} defined for a given general matrix *A* as follows: $(A^{H})_{ij} = (a_{ji})^{*}$.

conjugate number
: The conjugate of a complex number $z = a + bi$ is $z^{*} = a - bi$.

d
: When found as the first letter of routine names, `d` indicates the use of the double-precision real data type.

dot product	The number denoted $x \cdot y$ and defined for given vectors *x* and *y* as follows: $x \cdot y = \Sigma_i x_i y_i$. Here x_i and y_i stand for the *i*-th elements of *x* and *y*, respectively.
double precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers *x* such that $2.23*10^{-308} < \| x \| < 1.79*10^{308}$. For this data type, the machine precision ε is approximately 10^{-15}, which means that double-precision numbers usually contain no more than 15 significant decimal digits. For more information, refer to *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.
eigenvalue	*See* eigenvalue problem.
eigenvalue problem	A problem of finding non-zero vectors *x* and numbers λ (for a given square matrix *A*) such that $Ax = \lambda x$. Here the numbers λ are called the eigenvalues of the matrix *A* and the vectors *x* are called the eigenvectors of the matrix *A*.
eigenvector	*See* eigenvalue problem.
elementary reflector(Householder matrix)	Matrix of a general form $H = I - \tau v v^T$, where *v* is a column vector and τ is a scalar. In LAPACK elementary reflectors are used, for example, to represent the matrix *Q* in the *QR* factorization (the matrix *Q* is represented as a product of elementary reflectors).
factorization	Representation of a matrix as a product of matrices. *See also* Bunch-Kaufman factorization, Cholesky factorization, *LU* factorization, *LQ* factorization, *QR* factorization, Schur factorization.
FFTs	Abbreviation for Fast Fourier Transforms. *See*"Fourier Transform Functions".
full storage	A storage scheme allowing you to store matrices of any kind. A matrix *A* is stored in a two-dimensional array *a*, with the matrix element a_{ij} stored in the array element $a(i,j)$.
Hermitian matrix	A square matrix *A* that is equal to its conjugate matrix A^H. The conjugate A^H is defined as follows: $(A^H)_{ij} = (a_{ji})^*$.
I	*See* identity matrix.
identity matrix	A square matrix I whose diagonal elements are 1, and off-diagonal elements are 0. For any matrix *A*, $AI = A$ and $IA = A$.
i.i.d.	Independent Identically Distributed.
in-place	Qualifier of an operation. A function that performs its operation in-place takes its input from an array and returns its output to the same array.
Intel® oneAPI Math Kernel Library (oneMKL)	Abbreviation for Intel® oneAPI Math Kernel Library.
inverse matrix	The matrix denoted as A^{-1} and defined for a given square matrix *A* as follows: $AA^{-1} = A^{-1}A = I$. A^{-1} does not exist for singular matrices *A*.
LQ factorization	Representation of an *m*-by-*n* matrix *A* as $A = LQ$ or $A = (L\ 0)Q$. Here *Q* is an *n*-by-*n* orthogonal (unitary) matrix. For $m \le n$, *L* is an *m*-by-*m* lower triangular matrix with real diagonal elements; for $m > n$,



where L_1 is an n-by-n lower triangular matrix, and L_2 is a rectangular matrix.

LU factorization
: Representation of a general m-by-n matrix A as $A = PLU$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

machine precision
: The number ε determining the precision of the machine representation of real numbers. For Intel® architecture, the machine precision is approximately 10^{-7} for single-precision data, and approximately 10^{-15} for double-precision data. The precision also determines the number of significant decimal digits in the machine representation of real numbers. *See also* double precision and single precision.

MPI
: Message Passing Interface. This standard defines the user interface and functionality for a wide range of message-passing capabilities in parallel computing.

MPICH
: A freely available, portable implementation of MPI standard for message-passing libraries.

orthogonal matrix
: A real square matrix A whose transpose and inverse are equal, that is, $A^T = A^{-1}$, and therefore $AA^T = A^TA = I$. All eigenvalues of an orthogonal matrix have the absolute value 1.

packed storage
: A storage scheme allowing you to store symmetric, Hermitian, or triangular matrices more compactly. The upper or lower triangle of a matrix is packed by columns in a one-dimensional array.

PDF
: Probability Density Function. The function that determines probability distribution for univariate or multivariate continuous random variable X. The probability density function $f(x)$ is closely related with the cumulative distribution function $F(x)$.

  For univariate distribution the relation is

$$F(x) = \int_{-\infty}^{x} f(t)dt .$$

For multivariate distribution the relation is

$$F(x_1, x_2, \ldots, x_n) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} \ldots \int_{-\infty}^{x_n} f(t_1, t_2, \ldots, t_n) dt_1 dt_2 \ldots dt_n .$$

positive-definite matrix
: A square matrix *A* such that $Ax \cdot x > 0$ for any non-zero vector *x*. Here · denotes the dot product.

pseudorandom number generator
: A completely deterministic algorithm that imitates truly random sequences.

QR factorization
: Representation of an *m*-by-*n* matrix *A* as $A = QR$, where *Q* is an *m*-by-*m* orthogonal (unitary) matrix, and *R* is *n*-by-*n* upper triangular with real diagonal elements (if $m \geq n$) or trapezoidal (if $m < n$) matrix.

random stream
: An abstract source of independent identically distributed random numbers of uniform distribution. In this manual a random stream points to a structure that uniquely defines a random number sequence generated by a basic generator associated with a given random stream.

RNG
: Abbreviation for Random Number Generator. In this manual the term "random number generators" stands for pseudorandom number generators, that is, generators based on completely deterministic algorithms imitating truly random sequences.

Rectangular Full Packed (RFP) storage
: A storage scheme combining the full and packed storage schemes for the upper or lower triangle of the matrix. This combination enables using half of the full storage as packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels as the full storage.

s
: When found as the first letter of routine names, `s` indicates the use of the single-precision real data type.

ScaLAPACK
: Stands for Scalable Linear Algebra PACKage.

Schur factorization
: Representation of a square matrix *A* in the form $A = ZTZ^H$. Here T is an upper quasi-triangular matrix (for complex *A*, triangular matrix) called the Schur form of *A*; the matrix *Z* is orthogonal (for complex *A*, unitary). Columns of *Z* are called Schur vectors.

single precision
: A floating-point data type. On Intel® processors, this data type allows you to store real numbers *x* such that $1.18*10^{-38} < |x| < 3.40*10^{38}$. For this data type, the machine precision (ε) is approximately 10^{-7}, which means that single-precision numbers usually contain no more than 7 significant decimal digits. For more information, refer to *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

singular matrix	A matrix whose determinant is zero. If A is a singular matrix, the inverse A^{-1} does not exist, and the system of equations $Ax = b$ does not have a unique solution (that is, there exist no solutions or an infinite number of solutions).
singular value	The numbers defined for a given general matrix A as the eigenvalues of the matrix AA^H. *See also* SVD.
SMP	Abbreviation for Symmetric MultiProcessing. Intel® oneAPI Math Kernel Library (oneMKL) offers performance gains through parallelism provided by the SMP feature.
sparse BLAS	Routines performing basic vector operations on sparse vectors. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors. *See* BLAS.
sparse vectors	Vectors in which most of the components are zeros.
storage scheme	The way of storing matrices. *See* full storage, packed storage, and band storage.
SVD	Abbreviation for Singular Value Decomposition. *See also* Singular value decomposition section in "LAPACK Auxiliary and Utility Routines".
symmetric matrix	A square matrix A such that $a_{ij} = a_{ji}$.
transpose	The transpose of a given matrix A is a matrix A^T such that $(A^T)_{ij} = a_{ji}$ (rows of A become columns of A^T, and columns of A become rows of A^T).
trapezoidal matrix	A matrix A such that $A = (A_1 A_2)$, where A_1 is an upper triangular matrix, A_2 is a rectangular matrix.
triangular matrix	A matrix A is called an upper (lower) triangular matrix if all its subdiagonal elements (superdiagonal elements) are zeros. Thus, for an upper triangular matrix $a_{ij} = 0$ when $i > j$; for a lower triangular matrix $a_{ij} = 0$ when $i < j$.
tridiagonal matrix	A matrix whose non-zero elements are in three diagonals only: the leading diagonal, the first subdiagonal, and the first super-diagonal.
unitary matrix	A complex square matrix A whose conjugate and inverse are equal, that is, that is, $A^H = A^{-1}$, and therefore $AA^H = A^H A = I$. All eigenvalues of a unitary matrix have the absolute value 1.
VML	Abbreviation for Vector Mathematical Library. *See*"Vector Mathematical Functions".
VSL	Abbreviation for Vector Statistical Library. *See*"Statistical Functions".
z	When found as the first letter of routine names, `z` indicates the use of the double-precision complex data type.

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

Third Party Content

Intel® oneAPI Math Kernel Library includes content from several 3rd party sources that was originally governed by the licenses referenced below:

- Portions© Copyright 2001 Hewlett-Packard Development Company, L.P.
- Sections on the Linear Algebra PACKage (LAPACK) routines include derivative work portions that have been copyrighted:

  © 1991, 1992, and 1998 by The Numerical Algorithms Group, Ltd.
- Intel® oneAPI Math Kernel Library supports LAPACK 3.5 set of computational, driver, auxiliary and utility routines under the following license:

  Copyright © 1992-2011 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

  Copyright © 2000-2011 The University of California Berkeley. All rights reserved.

  Copyright © 2006-2012 The University of Colorado Denver. All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
  - Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL

DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The original versions of LAPACK from which that part of Intel® oneAPI Math Kernel Library was derived can be obtained from http://www.netlib.org/lapack/index.html. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

- The original versions of the Basic Linear Algebra Subprograms (BLAS) from which the respective part of Intel® oneAPI Math Kernel Library was derived can be obtained from http://www.netlib.org/blas/index.html.
- XBLAS is distributed under the following copyright:

  Copyright © 2008-2009 The University of California Berkeley. All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.

  - Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
- The original versions of the Basic Linear Algebra Communication Subprograms (BLACS) from which the respective part of Intel® oneAPI Math Kernel Library was derived can be obtained from http://www.netlib.org/blacs/index.html. The authors of BLACS are Jack Dongarra and R. Clint Whaley.
- The original versions of Scalable LAPACK (ScaLAPACK) from which the respective part of Intel® oneAPI Math Kernel Library was derived can be obtained from http://www.netlib.org/scalapack/index.html. The authors of ScaLAPACK are L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley.
- The original versions of the Parallel Basic Linear Algebra Subprograms (PBLAS) routines from which the respective part of Intel® oneAPI Math Kernel Library was derived can be obtained from http://www.netlib.org/scalapack/html/pblas_qref.html.
- PARDISO (PARallel DIrect SOlver)* in Intel® oneAPI Math Kernel Library was originally developed by the Department of Computer Science at the University of Basel (http://www.unibas.ch). It can be obtained at http://www.pardiso-project.org.
- The Extended Eigensolver functionality is based on the Feast solver package and is distributed under the following license:

  Copyright © 2009, The Regents of the University of Massachusetts, Amherst.
  Developed by E. Polizzi
  All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ''AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Some Fast Fourier Transform (FFT) functions in this release of Intel© oneAPI Math Kernel Library have been generated by the SPIRAL software generation system (http://www.spiral.net/) under license from Carnegie Mellon University. The authors of SPIRAL are Markus Puschel, Jose Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo.
- Open MPI is distributed under the New BSD license, listed below.

  Most files in this release are marked with the copyrights of the organizations who have edited them. The copyrights below are in no particular order and generally reflect members of the Open MPI core team who have contributed code to this release. The copyrights for code used under license from other parties are included in the corresponding files.

  Copyright © 2004-2010 The Trustees of Indiana University and Indiana University Research and Technology Corporation. All rights reserved.

  Copyright © 2004-2010 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

  Copyright © 2004-2010 High Performance Computing Center Stuttgart, University of Stuttgart. All rights reserved.

  Copyright © 2004-2008 The Regents of the University of California. All rights reserved.

  Copyright © 2006-2010 Los Alamos National Security, LLC. All rights reserved.

  Copyright © 2006-2010 Cisco Systems, Inc. All rights reserved.

  Copyright © 2006-2010 Voltaire, Inc. All rights reserved.

  Copyright © 2006-2011 Sandia National Laboratories. All rights reserved.

  Copyright © 2006-2010 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.

  Copyright © 2006-2010 The University of Houston. All rights reserved.

  Copyright © 2006-2009 Myricom, Inc. All rights reserved.

  Copyright © 2007-2008 UT-Battelle, LLC. All rights reserved.

  Copyright © 2007-2010 IBM Corporation. All rights reserved.

  Copyright © 1998-2005 Forschungszentrum Juelich, Juelich Supercomputing Centre, Federal Republic of Germany

  Copyright © 2005-2008 ZIH, TU Dresden, Federal Republic of Germany

- The Safe C Library is distributed under the following copyright:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

- HPL Copyright Notice and Licensing Terms

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.
  3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed at the University of Tennessee, Knoxville, Innovative Computing Laboratories.
  4. The name of the University, the name of the Laboratory, or the names of its contributors may not be used to endorse or promote products derived from this software without specific written permission.